Checking Session-Oriented Interactions between Web Services

Pavel Parizek[1], Jiri Adamek[1,2]

[1]Charles University in Prague, Faculty of Mathematics and Physics,
Department of Software Engineering, Distributed Systems Research Group
{parizek,adamek}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

[2]Academy of Sciences of the Czech Republic,
Institute of Computer Science

Abstract

Although web services are generally envisioned as being stateless, some of them are implicitly stateful. The reason is that the web services often work as front-ends to enterprise systems and are used in a session-oriented way by the clients. Contrary to the case of stateless services, for a stateful web service there exist constraints to the order in which the operations of the service may be invoked. However, specification of such constraints is not a standard part of a web service interface, and compliance with such constraints is not checked by the standard web service development tools. Therefore, we propose in this paper to extend a web service interface by a constraint definition that is based on behavior protocols. Also, we implemented a tool that checks whether a given BPEL code complies with the constraints of all stateful web services it communicates with. The key idea behind the tool is to translate the BPEL code into Java and then to check the Java program using Java PathFinder with behavior protocol extension.

Keywords: web services, BPEL, session-oriented interactions, behavior protocols, model checking

1. Introduction

Web service is a software system with a well-defined interface that is able to perform certain operations (specified in the interface) upon request in the form of a network message. Standard web service technologies are based on XML - web services communicate with their clients (e.g. other web services) via exchange of XML messages using the SOAP protocol, and their interfaces are defined in the WSDL [19] language.

To implement its WSDL interface, a web service can use other web services, and, optionally, add its own business logic. Borrowing the terminology from the field of software components ([3, 11]), we call a service that uses other web services as a *composite web service*, and the services used by a composite web service as *primitive web services*. The key difference between a composite service and primitive services is that the inner structure (the implementation) of a composite service is externally visible, while in the case of a primitive service it is not; nevertheless, both primitive and composite services have externally visible WSDL interfaces. A composite web service typically works as a business process that integrates several primitive services in a non-trivial way.

In general, a composite web service can be implemented in any programming language that provides an API for the web service technologies (e.g. Java and C#) - the only requirement is that the implementation conforms to the service's WSDL interface and uses the SOAP protocol for the communication. Nevertheless, the usage of languages like BPEL [18] for implementation of composite web services (e.g. business processes) has become very popular recently. Therefore, we focus on composite web services implemented in BPEL (*BPEL web services*) in this paper.

Web services are commonly envisioned as being stateless, in the sense that any two invocations of a single web service are independent - no state is preserved by the service between invocations and the clients have to supply all necessary context information in the request messages. However, in practice, some web services are implicitly stateful, since they work as front-ends to stateful resources (e.g. enterprise systems) and are used in a *session-oriented* way by the clients (e.g. BPEL web services). For a stateful web service, there typically exist constraints to the order in which the operations provided by the service may be invoked. As an example, consider an airline web service that works as a front-end to the on-line reservation system: a client of such a service has to invoke its operations in a specific sequence (e.g. `book` followed by `confirm`, and then by `pay`).

Nevertheless, a specification of the constraints on the order of the operation invocations is not a part of the web

service's WSDL interface and therefore the compliance with the constraints cannot be checked. An obvious idea is (i) to attach a formal specification of the constraints to a stateful primitive web service in addition to its WSDL interface and (ii) to create a tool for checking compliance of a composite web service with the constraints.

The contributions of this paper are:

(1) A technique for checking whether a BPEL web service is compliant with the specification of constraints on the order of invocations of every stateful primitive service it uses (those constraints are defined in the formalism of behavior protocols [16]), and the implementation of the technique in the BPEL checker tool [13].

(2) Evaluation of the technique on a case study; all examples in the paper are taken from the case study.

The remainder of the paper is organized as follows. Sect. 2 provides a short overview of WSDL and BPEL. Sect. 3 introduces behavior protocols and elaborates on the concept of compliance with the constraints specified via behavior protocols. Sect. 4 describes the proposed technique for checking whether a BPEL web service uses stateful primitive web services in compliance with the constraints, and Sect. 5 provides details about the evaluation of the technique on a case study. The rest of the paper contains related work and a conclusion.

2. WSDL and BPEL

The Web Services Description Language (WSDL) [19] is an XML-based language for definition of a web service interface. In particular, a WSDL document for a specific web service describes ports of the service, operations provided on the ports, and messages that represent input parameters and results of the operations. Each WSDL document can be divided into two parts: (i) the definition of data types (via XML Schema), abstract messages, operations and port types, and (ii) the deployment-related information like definition of endpoints and port instances. An endpoint definition specifies the binding of abstract messages and operations to a concrete network protocol and message format, and a port instance associates a network address with a binding. In order to be usable by a BPEL web service, a WSDL document also has to contain BPEL-specific definition of partner links. A partner link is a "connection" between a BPEL web service and a primitive web service, which may involve one or two roles associated with port types. For synchronous and one-way operations, one role is associated with a port type, while for asynchronous (callback) operations, there are two roles (each associated with a port type): one for the request from a client and the other for the callback invocation.

The following fragment of a WSDL document specifies a web service that allows to book a flight (denoted as `AirlineService` in the rest of the paper).

```
<definitions name="AirlineService">
  <portType name="BookPort">
    <operation name="IsAvailable">
      // declaration of input/output messages
    </operation>
    // definition of Book, Confirm and Cancel
  </portType>

  <partnerLinkType name="BookSvcLink">
    <role name="bookSvc"
        portType="air:BookPort"/>
  </partnerLinkType>
</definitions>
```

The definitions of data types and messages are omitted from the `AirlineService` WSDL document, since messages and their content are not relevant for the technique proposed in this paper. Note that the `BookSvcLink` partner link type involves one role, what implies that all operations on the `BookPort` port are synchronous or one-way.

As indicated in Sect. 1, the Business Process Execution Language (BPEL) [18] is an XML-based language for definition of business processes on the basis of the web service technologies (SOAP, WSDL, etc). A BPEL process is a composite web service implemented in BPEL (a BPEL web service) that interacts with several primitive services.

BPEL supports atomic sending and receiving of messages (i.e. invoking a web service operation and wait for a reply), and typical features of modern programming languages like data manipulation operations, control-flow structures (e.g. if-else, loops, and parallel execution) and fault-handling.

A typical BPEL process definition consists of many elements, the most important being:

(1) import of WSDL documents for primitive web services used by the BPEL process;

(2) definition of partner links (with types from the WSDL documents), where each partner link corresponds to one primitive service used by the process;

(3) declaration of variables for message content, with types corresponding to messages defined in WSDL;

(4) primary (top-most) activity of the process.

BPEL supports two kinds of activities - *basic* and *structured*. Basic activities are used for low-level tasks like message sending and receiving (XML elements `<invoke>`, `<receive>` and `<reply>`) and data manipulation (`<assign>`), while structured activities represent control-flow structures (e.g. `<flow>`, `<while>`, and `<if>`-`<else>`).

In this paper, the focus is on message exchange-related basic activities, which can be divided into two groups - those that cause a message to be sent (`<invoke>` and `<reply>`) and those that cause a message to be received (`<receive>` and `<invoke>`). The `<invoke>` activity supports two styles of communication: request-response and one-way. Each message represents an invocation of an operation or a reply, and is specific to a partner link name, port type and operation

name (correlations and service instances are ignored - see Sect. 4.2 for details).

A structured activity may contain recursively nested activities of both kinds and also nested scopes, where a scope is a part of a BPEL process definition that provides context for nested variable declarations, various handlers (e.g. for faults), and its primary activity. The top-most element of the process definition - the `<process>` XML element - is a special scope, denoted as a root context.

The following fragment of a BPEL process definition illustrates the key concepts of BPEL on the interaction between the `AirlineCustomer` process (i.e. a BPEL web service) and two instances of a primitive service with the `AirlineService` WSDL interface.

```
<process name="AirlineCustomer">
  <partnerLinks>
    <partnerLink name="BookSvcOne"
        partnerLinkType="air:BookSvcLink"
        partnerRole="bookSvc" />
    <partnerLink name="BookSvcTwo" ... />
  </partnerLinks>

  // declaration of variables for messages

  <sequence> // primary activity
    // message-content related operations
    <flow>
      <invoke partnerLink="BookSvcOne"
          portType="air:BookPort"
          operation="IsAvailable" ... />
     <invoke partnerLink="BookSvcTwo" .. />
    </flow>
    <if> // (1)
       <condition>...</condition> // flight
  available at BookSvcOne ?
      ...
      <invoke partnerLink="BookSvcOne"
          operation="Book" ... />
      <if>
       <condition>...</condition> // flight
is booked ?
        <invoke partnerLink="BookSvcOne"
            operation="Confirm" ... />
        <else>
          <invoke partnerLink="BookSvcOne"
              operation="Cancel" ... />
        </else>
      </if>
      <elseif>
       // as above (the if-branch), but for
BookSvcTwo
       // a call to Cancel is not specified
      </elseif>
    </if>
  </sequence>
</process>
```

The BPEL process first creates connections to two instances of a stateful primitive web service with the `AirlineService` WSDL interface over the `BookSvcOne` and `BookSvcTwo` partner links. Then it invokes the `IsAvailable` operation over both `BookSvcOne` and `BookSvcTwo` in parallel (via `<flow>`), and, depending on the availability of a flight, it invokes `Book` either over `BookSvcOne` or over `BookSvcTwo`. If `BookSvcOne` was selected, then the process invokes `Confirm` or `Cancel`, while if `BookSvcTwo` was selected, then it invokes `Confirm` or does nothing (call to `Cancel` over `BookSvcTwo` is not specified). Note that the `<else>` branch (i.e. the default option) of the top-most `<if>` activity (marked with (1) in the example) is not defined, so that if the flight won't be booked for one of `BookSvcOne` and `BookSvcTwo`, nothing happens.

3. Correct Usage of Primitive Web Services

In this section, the idea of primitive web service usage correctness is described in detail. The term is defined in the context of the following setup: let C be a composite web service implemented in BPEL, which uses stateful primitive web services P_1, ..., P_n. As emphasized in Sect. 1, the correctness here means that C behaves in compliance with the specification of constraints upon the order of operation invocations for each P_i.

To refine this definition, in Sect. 3.1 session protocols are introduced, specifying the correct orders of invocations of operations provided by primitive web services, and in Sect. 3.2 compliance of C with the session protocol of P_i is defined.

3.1. Session Protocols

In [16], *behavior protocols* were introduced as a formal language to specify behavior of software components. They were successfully used for the SOFA [16] and Fractal [9] component models. In this paper, a behavior protocol is used to specify constraints on the order of invocation of operations provided by a stateful primitive web service within the scope of a single session. Such a behavior protocol is called *session protocol* and is specific to a single partner link.

To define the syntax and semantics of session protocols, it is not necessary to extend the original behavior protocols by any new construct; in fact, just a subset of the behavior protocol features defined in [16] is needed. This section describes the subset.

Syntactically, a session protocol is an expression over a set of *message tokens*. A message token denotes either *sending* or *receiving* of a *message*, while a message may be either a *request* for a web service operation or a *response* to such request (for a one-way operation only the request exists). E.g., a service calling an operation first sends a request and then it receives a response, while the service providing the operation first receives a request and then it sends a response. The structure of a message token is as follows: it consists of a symbol denoting either sending (`!`) or receiving (`?`), followed by a port type, the dot symbol (`.`), an operation identifier, and the symbol denoting either request

(^) or response ($). A port type is typically prefixed by a namespace identifier, ":" is used as a delimiter. In this paper, a short version of namespace identifiers (a simple string) is used, which is not necessarily unique in general. Long version (fully qualified name) would consist of an URI, which may be quite complex, thus making the message tokens unreadable. The "NULL" identifier is special - it specifies an empty behavior.

While a message token expresses sending or receiving of a single message (the most simple behavior that can be specified), session protocol *operators* allow to express complex behavior, using the message tokens as building blocks. There are three operators: non-deterministic alternative (+), sequence (;) and repetition (*).

Finally, abbreviations are used as syntactic sugar for typical constructions: for an operation o, the symbol ?o stands for the sequence ?o^;!o$ and !o stands for !o^;?o$, and for an arbitrary session protocol P, the symbol ?o{P} stands for ?o^;P;!o$.

How the operators and abbreviations can be used is illustrated in the following example that corresponds to the BPEL code from Sect. 2 - the session protocol specifies the constraints upon the order of invocations of a primitive web service having the AirlineService WSDL interface.

```
?air:BookPort.IsAvailable ;
(
  (
    ?air:BookPort.Book ;
    (
      ?air:BookPort.Confirm^
      +
      ?air:BookPort.Cancel^
    )
  )
  +
  NULL
)
```

It states that first the synchronous IsAvailable operation on air:BookPort must be invoked (i.e. a request is received and later a response is sent), then an optional invocation of the synchronous Book operation follows, and if Book was invoked, it is required that one of the Confirm and Cancel one-way operations is invoked (i.e. a request is received, but no response is sent). The NULL branch is for the case when Book was not invoked.

Note that a session protocol is an abstraction of the primitive service's behavior, since only message exchange-related events (i.e. sending and receiving) and control-flow are modeled. Original behavior protocols abstract from method parameters that are analogous to message content, and thus session protocols do not model message content.

Moreover, we do not model parallelism in session protocols (although original behavior protocols [16] support it), since invocations of a specific primitive service in the scope of a single session are rarely performed in parallel; nevertheless, a BPEL web service can interact with several primitive services in different sessions in parallel.

3.2. Compliance with Session Protocols

To define the compliance of a composite web service C implemented in BPEL with a stateful primitive web service P_i equipped with a session protocol S_i, the semantics of session protocols has to be defined first. Formally, a session protocol is a regular expression upon the alphabet A_i of all message tokens that occur in S_i. Each sequence of the message tokens, that is specified by a session protocol, is called a *trace*. All traces that are specified by S_i form the *language* $L(S_i)$, providing the semantics of the protocol. Detailed definition of the language of a particular session protocol can be found in [16]. The language of the protocol from Sect. 3.1 consists of three traces; this is one of them:

```
?air:BookPort.IsAvailable^,
!air:BookPort.IsAvailable$,
?air:BookPort.Book^,
!air:BookPort.Book$,
?air:BookPort.Confirm^
```

When C is run, its real communication with P_i can be formalized as a sequence of message tokens from A_i as well. Let us denote a message received by P_i from C by a message token beginning with the '?' symbol, and a message sent by P_i to C by a token starting with '!'. For each run of C and P_i, the sequence of such tokens (ordered by the time in which they occurred) formalizes the communication between C and P_i during the run. Let us denote the set of all such sequences (for all possible real runs) as $L(C, P_i)$.

Based on the relation between the specified traces ($L(S_i)$) and the sequences denoting the communication during the real runs ($L(C, P_i)$), we formally define the compliance of a composite web service with a session protocol as follows:

Definition (*session compliance*). A composite web service C *complies* with the session protocol S_i of a primitive web service P_i iff $L(C, P_i) \subseteq L(S_i)$.

E.g., the BPEL code from Sect. 2 is not compliant with the session protocol from Sect. 3.1 in case of the primitive service connected over the BookSvcTwo partner link (a call to Cancel is missing). The trace specified in the BPEL code, but not present in the language of the protocol, is:

```
?air:BookPort.IsAvailable^,
!air:BookPort.IsAvailable$,
?air:BookPort.Book^,
!air:BookPort.Book$
```

4. Checking BPEL against Session Protocols

The proposed technique aims at checking the property of session compliance between a composite web service implemented in a limited version of the BPEL language

(details in Sect. 4.1) and session protocols of stateful primitive web services, as defined in Sect. 3.2. As there exists no model checker for BPEL that could be changed or customized to check the session compliance, we decided to use the following two-step process:

(1) Translation of BPEL implementation of a composite web service into a Java program using modified version of the B2J tool (details in Sect. 4.2).

(2) Checking of the Java program with a tool based on combination of the Java PathFinder model checker and behavior protocol checker (details in Sect. 4.3).

We decided to use this approach (instead of implementing our own model checker for BPEL) because reuse of a well-established model checker is a common practice in the area of formal verification. The main advantage is that such a model checker supports complex heuristics and optimizations; implementing those in our own tool would be very difficult and time consuming.

We decided to use the Java PathFinder model checker (JPF) [17] for two reasons: first, it has Java as its input language; Java is very rich and therefore it is possible to transform BPEL to Java with minimal loss of information caused by language difference (however, we remove some information on purpose as an optimization of checking). Second, we already had a working JPF-based solution for checking of correspondence between Java code and behavior protocols of software components [15] that we reused with only a minor modification.

4.1. Level of BPEL Support

An input of the first step (BPEL to Java translation) is the implementation of a composite web service in the limited version of the WS-BPEL 2.0 language [18], which is the most recent standardized version of BPEL. The unsupported features of WS-BPEL 2.0 include:

(i) time-related activities (e.g. `onAlarm` event, `<wait>` activity, etc), since our model checking tool for Java (i.e. JPF) does not handle time properly,

(ii) handling of faults (e.g. network-related errors), and

(iii) deployment-related information (endpoints and bindings), since they are necessary only at run-time - for the purpose of the proposed checking technique, a message source and destination is uniquely identified by a partner link name, port type and operation name.

Technically, if an unsupported BPEL construct is found in the input (BPEL code), it is ignored and a warning is printed.

4.2. Translation of BPEL into Java

As indicated above, checking of compliance between BPEL implementation of a composite web service and session protocols of stateful primitive services involves translation of BPEL code into a Java program. The principal requirement upon BPEL-to-Java translation is that it has to preserve important aspects for compliance checking, i.e. message exchange-related activities and control-flow structures, while all the other aspects (e.g. message content) should be ignored in order to make the Java program as simple as possible.

Stemming from this requirement, the key ideas of our approach to translation of BPEL code into a Java program are (a) to represent message exchange-related basic activities of BPEL as calls to special methods in Java, (b) to perform complete abstraction of message content (XML) and data manipulation operations, and (c) to map each structured activity of BPEL on its natural counterpart in Java. For the purpose of translation, a modified version of the B2J tool [1], which is a part of the Eclipse STP project, is used.

Complete abstraction of message content (XML data) and data-manipulation operations (like assignment and XPath expressions) is performed, since session protocols model only control-flow and message exchange-related events (i.e. no data), and, moreover, inclusion of the code for manipulation with XML data in the Java program would cause a significant growth of the JPF state space size. Technically, XML data-related constructs of BPEL are either ignored or modeled via non-determinism (using the `Verify` class provided by JPF). The former applies, e.g., to the `<assign>` activity and the latter to XPath expressions. Correlations are also ignored in our checking algorithm, since there is no need to distinguish between several instances of the same primitive service that interact with the composite service over partner links with the same name; inherent assumption is that interaction between a composite web service and a specific instance of a primitive service can be uniquely identified by a partner link name.

As to (a) above, each message exchange related basic activity of BPEL is translated into a call of a specific Java method. Specifically, activities that cause a message to be sent (i.e. `<invoke>` and `<reply>`) are translated into calls of the `stubSEND` method and activities that cause a message to be received (i.e. `<receive>` and `<invoke>`) are translated into calls of the `stubRECEIVE` method. Note that for each `<invoke>` in the BPEL code, there are calls to both `stubSEND` and `stubRECEIVE` only if an output variable is specified in the activity (i.e. a reply is expected), otherwise there is only a call to `stubSEND`. The `stubSEND` and `stubRECEIVE` methods are empty stubs that are added to the Java program by the translator tool. As parameters, both methods get a partner link name, a port type, and an operation name; therefore, each call to one of these methods uniquely identifies the corresponding message for the purpose of compliance checking.

Consequence of (i) modeling message submission and reception via calls to empty Java methods, (ii) no support for fault-handling, and (iii) missing support of time in JPF is that complete abstraction of the behavior of network communication infrastructure is performed during translation

of BPEL into Java. In particular, all message exchange-related activities are considered to finish immediately with a success; e.g., for the `<receive>` activity, the message to be received is considered to be already available at the time the activity is executed (there is no point in modeling a wait for a message if JPF has no support for time).

Other basic activities (besides `<invoke>`, `<receive>` and `<reply>`) are either mapped to a natural Java counterpart (e.g. `<exit>` is translated to the `System.exit(0);` statement) or ignored, resp. not supported. The latter applies to the `<assign>`, `<wait>` and `<empty>` activities.

Structured activities are, in general, mapped to their natural counterparts in the Java language. For illustration, `<flow>` is mapped to a set of parallel threads and `<while>` is mapped to a loop. Details on the translation of selected structured activities (not illustrated on the example below) and further examples are in [14].

All the other possible elements of a scope (i.e. sub-elements of `<scope>` or `<process>`), like fault handlers and declaration of variables, are either ignored or mapped onto their Java counterparts in a very similar way to some other BPEL construct (e.g., event handlers are translated in almost the same way as the `<pick>` activity).

For illustration, a fragment of the Java code generated by translation of the BPEL code from Sect. 2 follows (more complete version is in [14]):

```
public void activity1() {
  activity8(); // <flow>
  activity11(); // <if>
}

public void activity8() { // <flow> activity
  Thread th1 = new Thread() {
    public void run() { activity9(); }
  };
  Thread th2 = new Thread() { ... };
  // threads are started via Thread.start
  // wait till the threads finish (via calls of Thread.join)
}

public void activity9() {// synch. <invoke>
  stubSEND("BookSvcOne",
      "air:BookPort", "IsAvailable");
  stubRECEIVE("BookSvcOne",
      "air:BookPort", "IsAvailable");
}

public void activity11() { // <if> activity
  int if3 = Verify.random(2);

  if (if3 == 0) { /* if branch */ }
  if (if3 == 1) { /* else-if branch */ }
  if (if3 == 2) { /* for missing else */ }
}
```

4.3. Cooperation of Java PathFinder with Behavior Protocol Checkers

As indicated above, the proposed technique aims at checking the property of session compliance between a composite web service implemented in BPEL and session protocols of several primitive services. For the Java program generated from BPEL code during translation, the original property (related to BPEL) can be rephrased as compliance of all possible sequences of calls to the `stubSEND` and `stubRECEIVE` methods with session protocols of the primitive services.

For checking of a Java program against the given property, an extension of the approach presented in [15] is used. The key idea of [15] is to check compliance of a Java program to a behavior protocol via combination of JPF with behavior protocol checker (BPC) [10]. Specifically, both checkers cooperate during traversal of their state spaces: JPF notifies BPC on execution of important Java byte code instructions that correspond to behavior protocol events (method invocations and returns in case of [15]), while traversing the state space of a Java program, and BPC checks whether the events form a trace allowed by the behavior protocol. If JPF notifies BPC about an event that is not allowed at the particular point in the behavior protocol's state space, a violation of compliance between Java code and behavior protocol is reported. Moreover, JPF can backtrack only if BPC agrees so that each trace in the behavior protocol is checked completely (coordination of backtracking is used).

Here the approach of [15] is reused with the following modifications:

(i) Since a composite web service (implemented in BPEL) can interact with several stateful primitive services, JPF cooperates with several instances of BPC in parallel, where each BPC instance corresponds to a single primitive service instance "connected" via a partner link and is therefore identified by the partner link's name.

(ii) Important byte code instructions with respect to session protocols are calls of the `stubSEND` and `stubRECEIVE` methods. When JPF detects a call of one of these methods, it extracts the values of the method's parameters (partner link name, port type and operation), and notifies the BPC instance associated with the partner link name about the particular event (e.g., `?<port type>.<operation>^` for a call to the `stubSEND` method).

(iii) JPF is allowed to backtrack only if all instances of BPC agree; in a similar way, JPF and all BPC instances have to reach an end state at the same time.

5. Evaluation

We have created a prototype implementation of the proposed technique in the BPEL checker tool [13] and applied it to a case study (Fig. 1), which models a customer

process for booking a flight at an airline service and paying for the flight via a credit card. Specifically, the case study involves the `AirlineCustomer` business process (a fragment of its BPEL definition is in Sect. 2) and five primitive web services (`Airline Service 1`, `Airline Service 2`, `Payment Terminal`, `Card Center 1`, `Card Center 2`).

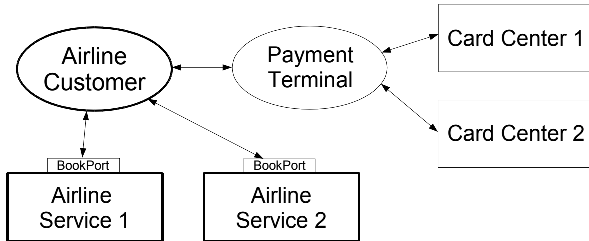

Figure 1: Architecture of the case study

Let us focus on the interaction between `AirlineCustomer` in the role of a composite web service and `Airline Service 1+2` in the roles of stateful primitive services (pointed to by the `BookSvcOne` and `BookSvcTwo` partner links from `AirlineCustomer`). A fragment of the WSDL interface for these primitive services is presented in Sect. 2 (`AirlineService`) and their session protocol is introduced in Sect. 3.1. The BPEL checker successfully detected a violation of the session protocol in the interaction between the `AirlineCustomer` process and `Airline Service 2` (connected to the process via the `BookSvcTwo` partner link) - the missing call to `Cancel` (end of Sect. 2) was reported.

A consequence of the complete abstraction of the message content (XML data) and data-manipulation operations (including XPath expressions), performed during the BPEL to Java translation (Sect. 4.2), is that spurious errors may be reported by the BPEL checker. However, if the abstraction would not be employed, checking of session compliance between BPEL code and session protocols might be infeasible in some cases.

One of the typical cases of a spurious error is depicted on the following example that involves a slightly modified fragment of the BPEL code from Sect. 2 (`AirlineCustomer`).

```
<if>
  <condition>expr</condition>
  <invoke partnerLink="BookSvcOne"
      operation="Confirm" ... />
</if>
<if>
  <condition>not expr</condition>
  <invoke partnerLink="BookSvcOne"
      operation="Cancel" ... />
</if>
```

The difference from the original BPEL code in Sect. 2 is that a separate `<if>` activity with an inverted condition is used to guard the call to `Cancel`, instead of an `<else>` branch. As indicated in Sect. 4.2, an `<if>` activity is translated to a non-deterministic choice between several alternatives, including one for the case when the condition associated with `<if>` is satisfied and one for the "default" case (with all conditions - for `<if>` and all `<elseif>` - not satisfied). Thus, when checking the BPEL code fragment above against the protocol `?air:BookPort.Confirm^ + ?air:BookPort.Cancel^` (a fragment of the session protocol from Sect. 3.1), the tool may report a spurious error that involves a call to `Confirm` followed by a call to `Cancel` (if the alternative for the case of satisfied condition is selected for both `<if>`s).

Since the BPEL checker does not check whether a detected error is real or possibly spurious (e.g. via simulation run driven by the error trace) and also does not provide any hints with respect to this issue, it is up to the user to manually examine each reported error in order to find whether it is spurious or real. Addressing this issue is a future work.

6. Related work

There exist several approaches to behavior verification of web services implemented in BPEL that aim at checking various properties of interactions among web services. However, as far as we know, none of them aims at checking of session compliance as the technique proposed in this paper. Short characteristic of selected approaches is provided below.

In [7], the authors present a technique for checking LTL properties of multiple interacting web services implemented in BPEL. The key idea of the technique is (i) translation of BPEL implementation of each web service into a process specified in Promela (input language of the SPIN model checker [8]), so that a complete Promela model is created, and (ii) checking of the Promela model against LTL properties using SPIN. The technique performs no abstraction of message content and data manipulation operations (a subset of XML Schema and XPath is translated into Promela as well), and supports message content-related properties.

The technique presented in [6] aims at checking of correspondence between implementation (in BPEL) and specification (in Message Sequence Charts - MSC) of a composite web service with respect to the traces of sent and received messages during interaction with (primitive) web services. The key idea is the translation of both BPEL implementation and MSC specification into the Finite State Process (FSP) notation, and subsequent checking of trace equivalence between FSP models of BPEL and MSC using the LTSA model checker [12].

There are also approaches to service behavior modeling that try to address sessions explicitly. SCC [2] provides the

user with powerful mechanisms of session naming and scoping, inspired by π-calculus. The sessions here allow to specify complex interactions between the processes (not only the classic procedure calls). In addition, sessions may be closed; therefore, interruption, service cancellation, or update can be specified. SOCK [4] is a formal framework for service communication and composition. It consists of three specification languages - the service behavior calculus, the service engine calculus, and the service system calculus. Those three languages form three layers of the framework; sessions are supported on all the layers.

Although both SCC and SOCK are languages for service specification explicitly dealing with sessions, no tools were developed so far for verification of SOCK or SCC specifications. The purpose of the languages is different: to provide a compact process-algebra-like formal language for the domain of services allowing theoretical analysis of services, and also direct interpretation of code written in the languages as a part of a service implementation.

7. Conclusion and Future Work

In this paper, we presented a technique for checking session compliance - the compliance of a composite web service implemented in BPEL with the constraints on the order of operation invocations on primitive web services (specified via session protocols). The key idea of the technique is to translate the BPEL implementation of a composite web service into Java and then to check the Java program with a tool based on combination of Java PathFinder and the behavior protocol checker. We have implemented the proposed technique in the BPEL checker tool [13] and illustrated usability of the tool for detection of violations of the session compliance on a case study.

The main issue of the technique is that spurious errors may be reported by the checking tool. This is a consequence of the abstraction of message content (XML data) and data manipulation operations during the BPEL to Java translation. As a future work, we plan to address this issue via partial abstraction of XML data or the well-known approach of counterexample guided abstraction refinement (CEGAR) [5].

Acknowledgments

This work was partially by the Czech Academy of Sciences project 1ET400300504 and partially supported by the ITEA/EUREKA project OSIRIS Σ!2023.

References

[1] B2J: tool for BPEL to Java translation, http://www.eclipse.org/stp/b2j

[2] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro: SCC: a service centered calculus, Proceedings of WS-FM 2006, LNCS 4184

[3] T. Bures, P. Hnetynka, and F. Plasil: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proc. of SERA 2006, IEEE CS

[4] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro: SOCK: a calculus for service oriented computing, Proceedings of ICSOC 2006, LNCS 4294

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith: Counterexample-guided abstraction refinement, Proc. of 12th CAV, LNCS 1855, 2000

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer: Model-based Verification of Web Service Composition, Proceedings of ASE'03, IEEE CS

[7] X. Fu, T. Bultan, and J. Su: Analysis of Interacting BPEL Web Services, Proceedings of WWW'04, ACM

[8] G. Holzmann: The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003, http://www.spinroot.com

[9] P. Jezek, J. Kofron, and F. Plasil: Model Checking of Component Behavior Specification: A Real Life Experience, Proceedings of FACS'05, ENTCS, vol. 160

[10] M. Mach, F. Plasil, and J. Kofron: Behavior Protocol Verification: Fighting State Explosion, IJCIS, Vol. 6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, 2005

[11] J. Magee, J. Kramer: Dynamic Structure in Software Architectures, Proceedings of FSE'4, Oct 1996

[12] J. Magee, J. Kramer: Concurrency - State Models and Java Programs, John Wiley, 1999, http://www.doc.ic.ac.uk/ltsa/

[13] P. Parizek: BPEL checker, 2007 http://dsrg.mff.cuni.cz/projects.phtml?p=bpelchecker

[14] P. Parizek, J. Adamek: Modeling and Verification of Session-Oriented Interactions between Web Services: Compliance of BPEL with Session Protocols, Tech. Report No. 2008/2, Dep. of SW Engineering, Charles University, Jan 2008

[15] P. Parizek, F. Plasil, and J. Kofron: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker, Proceedings of SEW'06, IEEE CS

[16] F. Plasil, S. Visnovsky: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda: Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, Apr 2003

[18] Web Services Business Process Execution Language (WS-BPEL), Version 2.0, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[19] Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl