Verification of Web Service Conversations Specified in WSCL

Zhifeng Gu*, Juanzi Li*, Jie Tang*, Bin Xu*, Ruobo Huang†
*DCST, Tsinghua Univ., Beijing, 100084, China
Email: {gzf, ljz, tangjie, xubin}@keg.cs.tsinghua.edu.cn
†CSDL, IBM, Beijing, 100085, China
Email: huangrb@cn.ibm.com

Abstract— **This paper studies one of the standards about web service conversations, the WSCL specification. We propose a method to verify conversations in WSCL. In this method we first propose using a translator to convert WSCL documents into promela, the modeling language of the model checker SPIN. Then, we run SPIN to check the conversation model against the correctness properties specified by the designer. A toolkit for WSCL verification is introduced at the end of this paper.**

I. Introduction

Web service has gained a lot of attention since 2000. Many standards have been proposed to enrich the web services stack [1], among which WSDL is one of the most influential standards.

WSDL defines the basic elements of a service, including operations, input/output messages and exceptions. In WSDL, there are four patterns defined for message exchanges; however, these patterns are too primitive to support complicated applications directly. The service interfaces specified in WSDL are unsustainable when the conversation participants are going to tailor their needs and offers according to the prevailing context or to coordinate multiple services in open and realistic environments [2]. Therefore, the complexities in open and realistic environments call for sustainable conversation protocols.

WSCL, a conversation definition language, is a W3C submission from the Hewlett-Packard Company. We adopt the model checker SPIN to verify the conversations specified in WSCL. We first give a scheme for translation from WSCL documents to promela, the modeling language of SPIN. Then, we write properties (temporal claim) and run SPIN to verify the correctness of the conversation. In real world design, the conversation specification and properties should be written independently, which will be helpful to find potential bugs.

The rest of this paper is organized as follows. In section II we review some existing standards and works about web service conversations, and give some comparisons with our work. Section III defines the scheme for translation from WSCL to promela, and discusses how to use SPIN to verify the correctness of a conversation. A toolkit implementing the translator and providing some other features is introduced in section IV. Finally, in section V we draw a conclusion and discuss the future work.

II. Related Work

The web service community has partially acknowledged that the interface description of WSDL is too simple to describe complex e-commerce services in real world applications [3].

Bultan [4] and Fu [5] study the conversation protocol from the perspective of the global behaviors. In their model, the message exchanges between different peers (services) are listened and recorded by a global watcher. The message queue recorded by the global watcher is called a conversation. The correctness of the conversation can be verified using LTL (Linear Temporal Logic) formulas.

More works focus on the local behaviors of an interface. Beyer [6] describes web service with the concept of $action$, which is a pair of a method and its outcome. Three kinds of interface constrains are defined within his work, which are signature constrains, consistency constrains and protocol constrains. For each type of the constrains, the author gives the algorithm for compatibility checking and substitutivity checking. In the traditional research area of software components, there are a lot of works on interface enhancement, one of which is the protocol specification of component interfaces [7].

Conversational issues have also been tackled in the area of multi-agent system. In fact, we think web service conversation and agent interaction share the same essentials, although their focuses on high level goals are different. Endriss introduces different levels of conformance as basic notions to check and enforce that the behavior of an agent is adopted to a public protocol regulating the interaction in a multi-agent system [8].

Generally speaking, a conversation protocol is part of service choreography. A choreography is a abstract specification of how a set of services works, while a conversation protocol specified in WSCL defines the interactions between two services. In the web services stack, there are two specifications for service choreography, WSCI and WS-CDL. WSCI is a W3C submission from BEA, BPMI, Oracle, SAP and etc. As the successor of WSCI, WS-CDL is a W3C working draft, and it may become a recommendation in the near future.

Service choreography is now a hot topic in the web service research community. Baldoni [9] studies how to check if the interactive behavior of a service respects the interaction schema (e.g. a choreography or an interaction protocol) that the services should follow. Zhao [10] proposes a small language CDL as a formal model of the simplified WS-CDL. In his work, he also uses SPIN to verify the correctness of a given choreography.

As a more powerful standard, WS-CDL can be applied to more situations than WSCL; however, like most standards in the web services stack, WS-CDL is verbose and complex [11], so it is not suitable to specify simple choreography with WS-CDL. A light weighted choreography standard may be needed to meet simple needs. WSCL is definitely one of the candidates.

As a summary, conversation protocol is usually modeled in a FSM-like style, and model checking is widely used to check the correctness of conversation protocols. Compared with existing works, our work is, as far as we know, the first attempt to verify the conversations specified in WSCL.

SPIN is a very popular model checker for the formal verification of distributed software systems. Many works adopt SPIN as the verification tool. For example, Kazhamiakin [12] describes a novel approach for the formal specification and verification of distributed processes in a web service framework, and exploits SPIN to perform V&V tasks. WSAT [13] uses SPIN to verify LTL properties against process models specified in BPEL4WS. Ankolekar [14] translates OWL-S process model into promela, and then uses SPIN to verify the process model.

III. Our Approach

A. Introduction to WSCL

A WSCL conversation specification consists of two lists: the interaction list and the transition list. The two lists contain definitions of interactions and transition rules respectively. An *interaction* may be one of the following five types:

- *Receive*: contains one *InboundXMLDocument*
- *Send*: contains one *OutboundXMLDocument*
- *ReceiveSend*: contains one *InboundXMLDocument*, one or more *OutboundXMLDocument*
- *SendReceive*: contains one *OutboundXMLDocument*, one or more *InboundXMLDocument*
- *Empty*: does not contain any document exchanged, but is used only for modeling the start and end of a conversation.

A *transition* contains one source interaction, one destination interaction, and optionally, a conditional message of this transition rule. The *initialInteraction* and *finalInteraction* attributes must be specified for a conversation. Section 2.6 of the WSCL specification talks about what a well-formed WSCL document should be.

Before introducing our translation scheme, we give a simple example, illustrated in figure 1. In the figure, each node (rectangle or circle) represents an interaction; and each directed edge represents a transition. The double circle nodes are initial and final interaction; the circle node is *Empty* interaction; and the rectangle nodes are *ReceiveSend* interaction.

In the conversation in figure 1, the client first initiates the login request. If the login request is valid, the conversation enters a normal request-response cycle. Finally, the conversation will be terminated by a logout request. This is the most widely used conversation pattern in today's web applications. The WSCL source code of this sample can be found in our WSCL toolkit.

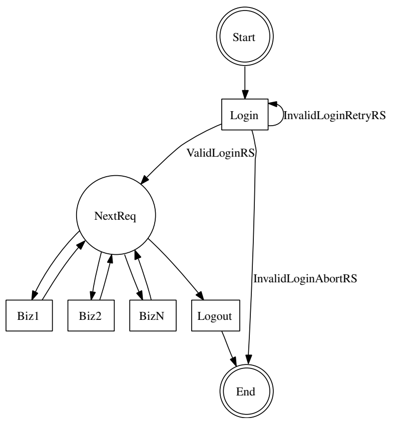

Fig. 1. A WSCL Sample

We need note that, in this sample, the *NextReq* interaction is *Empty*, which is not valid according to the WSCL specification. However, it is indeed helpful here to use the *Empty* interaction, because if the *NextReq* interaction is removed, we would have to define transition rules for each pair of all the interactions: Biz1, Biz2, ..., BizN. This becomes horrible when N gets large.

B. Translation from WSCL to Promela

WSCL defines the message exchange sequences between two and only two participants, so a natural idea is that, in the promela model, we could use two processes to represent the service provider and the service consumer, as shown in figure 2(a). Then, the conversation can be simulated through communications between these two processes. However, this simple model gets unsustainable when the participants cannot independently determine which transition to take (this will be explained in detail below). To solve this problem, we add a coordinator process. As shown in figure 2(b), the Client process and the Server process execute the conversation. When they cannot independently select a transition, they will ask the Coordinator process for coordination.

Our translation scheme is designed to handle any WSCL document that satisfies:

- It conforms to the WSCL schema.
- There is no transition whose destination is the *initialInteraction*.
- There is no transition whose source is the *finalInteraction*.
- There are no duplicated transitions (transitions that have the same source interactions and destination interactions).
- There are no duplicated IDs (e.g. message ID, interaction ID).
- All the IDs referred to in the document must be defined.

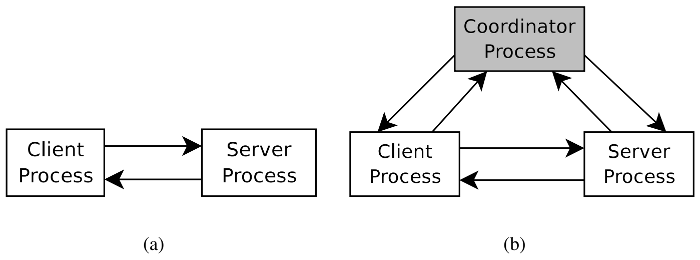

Fig. 2. The process model in promela

In fact. these restrictions are very loose. According to our experiences, nearly any well-formed WSDL document can be accepted by our translator.

In our scheme, each interaction is translated into a promela code block labeled with the ID of this interaction. Figure 3 gives the view of an interaction from the perspective of our translator. In the figure, the rectangle nodes are regular *Receive* and *Send* interactions; and the circle nodes are *Empty* interactions. The label on each node is the ID of the interaction. The directed edges represent the transitions associated with these interactions. An edge may have a label associated with it if it connects to a rectangle node or the *finalInteraction*. The label has three possible forms: *S:M*, *R:M* and *T*, which means the transition will send a message M, receive a message M and terminate the conversation respectively. These elements form a directed graph that the translator needs to deal with.

SendReceive and *ReceiveSend* interactions are not considered here, since they can be decomposed into several *Send* and *Receive* interactions as shown in figure 4. Note that transitions associated with a *ReceiveSend* (or *SendReceive*) interaction may have an optional element, *SourceInteractionCondition*. According to the value of this element, the transitions can be aggregated into several sets. For example, in figure 4, *C=M1* is a set of transitions whose *SourceInteractionCondition* is *M1*. A special set is *C=null*, which contains the transitions that do not have *SourceInteractionCondition* specified. The transitions in *C=null* are shared by all the decomposed *Send* nodes whose *OutboundXMLDocument* is not in the *SourceInteractionCondition* list; and this is done by adding an empty node.

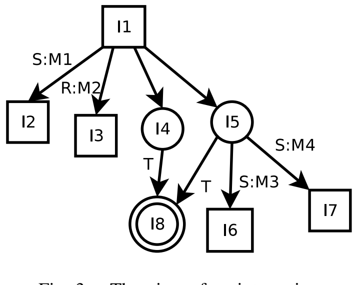

Fig. 3. The view of an interaction

In figure 3, the top interaction is the interaction to be translated. The interactions that connect to the top interaction

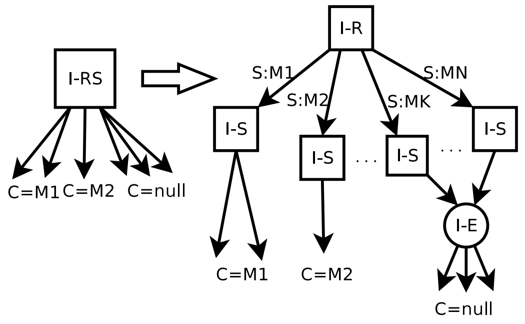

Fig. 4. Decompose a *ReceiveSend* interaction

directly are called direct successors. Correspondingly, the interactions introduced through the *Empty* interactions are called indirect successors. When the conversation runs into interaction N, the Server process and the Client process should make an identical selection among the direct and indirect successors of interaction N, and then they jump into the code block of the selected interaction.

1) Transition Ambiguity Elimination: To make the conversation executable, two problems need to be solved. One is, as mentioned above, the coordination of the Client and Server processes; the other is the ambiguity of transitions. First, we will talk about the second problem.

As shown in figure 5, the top interaction has two direct *Receive* successors and one indirect *Receive* successor. All of these successors have M as their inbound messages. Then, when a message M arrives, which transition should be selected? The WSCL specification does not address this problem clearly. Neither such condition is forbidden, nor it gives a solution. We can say such a conversation design is ambiguous and need to be revised; however, we think, on the other side, this problem is very common when we try to re-use existing conversation designs as sub-conversations. If we define some rules that can eliminate the ambiguity, it would become a mechanism to modularize and re-use conversation designs. In our work, we define the following rules to uniquely select an effective transition from a set of transitions that conflict with each other.

1) The type of the destination interaction. Transitions lead-

ing to non-*Empty* interaction will be processed first.

2) The literal order of transitions in WSCL document. The first specified transition will be processed first.
3) Indirect successors will be expanded in a deep first order.

According to these rules, the transition processed first will be selected as the effective transition. Regarding the case in figure 5, the transitions leading to the direct successors will be processed first. If the left most transition is specified before the other one, then it is the effective transition.

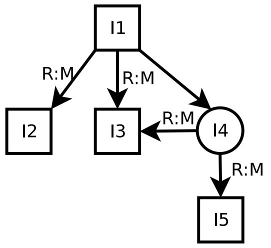

Fig. 5. The ambiguity of transitions

2) Execution Coordination: Next, we will discuss the coordination between the Client process and the Server process. The goal of the coordination task is to make these two processes always making the same selection when multiple transitions are available. Back to figure 3, if all the direct and indirect successors have the same type, i.e. *Send* or *Receive*, then the coordination task gets easy. The transition for next step can be selected independently by one of the participants whose successors are all *Send*. In our transition scheme, the sender will select a message randomly from all possible messages and select the transition according to the outgoing message. Correspondingly, the receiver should wait for an message and select the transition according to the incoming message. We say a WSCL conversation is self-contained, if, for each interaction in the conversation, all the successors of the interaction have the same type.

When the types of the successors become mixed, which means the successors contains both *Send* and *Receive* interactions, the coordination task gets a little complex. None of the participants can make a selection independently now. It must be determined first that who is the sender and who is the receiver. In real world application, this might be determined by the semantics carried by the payloads. However, in the conversation itself, there is no such information, so we add a coordinator process to solve the problem. When the Client and the Server process find its subsequent interactions containing both *Send* and *Receive* interactions, they will send a message to the Coordinator process. The Coordinator will then assign an action to the Client process and the Service process, which guarantees that one is *send*, and the other is *receive*. Then the process, which has been assigned the *send* action, will decide which message to send as discussed in the previous paragraph. We say such a WSCL design is not self-contained.

Comparing these two different types of WSCL design, we think the self-contained design is better, since its execution does not depend on any semantics outside the design itself. In real world applications, there will not be a coordinator that always gives consistent decisions. Regarding non-self-contained design, if the conversation participants have different understanding on the data semantics carried by the payloads, it is likely to cause a failure on the conversation layer due to the mismatched selection of transitions.

Another problem needs to be concerned is the termination of the conversation. A conversation is terminated when it reaches the *finalInteraction*. When the execution of a conversation runs into an interaction that has *finalInteraction* as its direct or indirect successor, the Client process and the Server process must agree on whether they should continue the conversation or not. This is also a coordination issue. To solve this problem, the Coordinator process must be able to support another action: *terminate*. As a summary, the Coordinator may give three possible instructions:

- Server process: *send*, Client process: *receive*
- Server process: *receive*, Client process: *send*
- Server process: *terminate*, Client process: *terminate*

3) The Code Skeletons: Here we will have a look at the code skeletons used in our translator. The code skeleton for sender and receiver is shown in listing 1 and listing 2 respectively. As discussed above, the sender will first pick up a message randomly and send it to the receiver. The rest of the code is to select a transition according to the message ID. This part of the sender and the receiver is completely the same.

These two code snippets are for self-contained WSCL designs only. Due to the limitation of the space, the code skeletons for non-self-contained WSCL designs are not listed here. Please refer to the output of our toolkit.

```
InteractionId:
  if
     :: var = value1;
     :: var = value2;
     :: ...
     :: var = valueN;
  fi;
  OUT ! var;
  if
    :: var == value1 -> goto direct_successor1;
    :: var == value2 -> goto direct_successor2;
    :: ...
    :: var == value11 ||
       var == value12 || ... ->
              goto empty_successor1;
    :: var == value21 ||
       var == value22 || ... ->
              goto empty_successor2;
  fi;
```

Listing 1. Code skeleton for sender

4) Counterpart Generation: The last step is to generate the counterpart document from the original conversation document. WSCL specifies the conversation from the viewpoint of one of the participants (usually the service provider), so we need to generate the conversation document of the counterpart process. It is easy to derive the counterpart document

```
InteractionId:
  IN ? var;
  if
    :: var == value1 -> goto direct_successor1;
    :: var == value2 -> goto direct_successor2;
    :: ...
    :: var == value11 ||
       var == value12 || ... ->
              goto empty_successor1;
    :: var == value21 ||
       var == value22 || ... ->
              goto empty_successor2;
  fi;
```

Listing 2. Code skeleton for receiver

from an existing WSCL document by inverting the message directions. In detail, the counterpart can be generated through the following steps.

1) Change *Send* to *Receive*, and accordingly revert the message direction from outbound to inbound.
2) Change *Receive* to *Send*, and revert the message direction as well.
3) Change *SendReceive* to *ReceiveSend*, and revert the message directions as well.
4) Change *ReceiveSend* to *SendReceive*, and revert the message directions as well.

C. Verification of the Conversation Model

So far, we have got the promela model for WSCL document. In this section, we will discuss how to verify the correctness of the model against a given set of properties using SPIN.

SPIN provides several ways to specify correctness properties, including basic assertion statements, end-state labels, progress-state labels, accept-state labels, never claims and trace assertions. In our work, we use two methods to express correctness properties. The fisrt one is LTL formulas. LTL formulas are not natively supported by the grammar of promela, but they can be translated into never claims by SPIN with the command line option "-f". The second is the trace assertions. A trace assertion expresses properties of message channels, and in particular it formalizes statements about valid or invalid sequences of operations that processes can perform on message channels [15]. Since WSCL is a specification all about message sequence, trace assertion is suitable for expressing properties of WSCL.

```
#define at_Login Server@Login
#define iLogin_Passed (iLogin_PAS_CNT > 0)
...
#define mLoginRQ_Sent (mLoginRQ_SND_CNT > 0)
#define mLoginRQ_Received (mLoginRQ_RCV_CNT > 0)
...
bit iLogin_PAS_CNT = 0;
...
bit mLoginRQ_SND_CNT = 0;
bit mLoginRQ_RCV_CNT = 0;
```

Listing 3. Predefined Expressions

In order to write LTL properties against the conversation, we predefine some expressions in the generated code. Listing 3 gives an example definition for interaction "Login" and message "LoginRQ". For each interaction, we define a bit variable "i + InteractionId + _PAS_CNT", and for each message, we define two variables "m + MessageId + _SND_CNT" and "m + MessageId + _RCV_CNT". The meaning of these predefined variables are self-explained by their names. Further, for convenience, we also define several macros as shown in listing 3. The special macro, "at + _InteractionId", means the conversation is currently running within the interaction whose ID is "InteractionId". With these assistant variables and macros, we can write LTL formulas like:

```
[] ! iNextReq_Passed ||
(! iNextReq_Passed U mValidLoginRS_Sent)
```

which means *NextReq* will never be entered if *ValidLoginRS* has not been sent. Anyway, writing LTL formula is a very skillful job. Matthew introduces many patterns for temporal properties in [16]. This is very helpful to specify complex properties.

Writing trace assertions is more straightforward than writing LTL formulas. In a trace assertion, the sending and receiving action does not actually send or receive a message. They just specify an action to be matched on the channel, so they are called events in a trace assertion. The control structures (if...fi and do...od) can also be used within trace assertions. They can describe branches and cycles in the message sequence. Using events and control structures, the designer can write the message sequence to be matched. An example of trace assertion can be found in our toolkit (see section IV).

As well as the properties specific to a conversation design, SPIN can also check some generic properties, such as deadlock freedom, termination, etc. In every WSDL design, unreachable interactions and termination of the conversation are two basic properties. They are usually violated by the absence of some necessary transitions. In our method, running SPIN on the bare model will check these two properties efficiently. SPIN will report the line numbers of unreachable states in promela code. Using these line numbers, we could trace back to the unreachable and pending interactions easily.

In principle, running SPIN on the bare model will also check the deadlock freedom property of the WSCL design, but, in our method, it is impossible to have deadlocks in the model. As explained in previous sub-section, the Server process and the Client process are completely symmetric in the generated model. As the channel never discards messages, the message sequence will always match between these two processes. Deadlock detection might be useful if the designer is going to make a sub-conversation of the original design. In this case, the Server process and the Client process are all human-designed, and may contain deadlocks.

IV. Implementation: A Toolkit for WSCL

According to the content in the previous section, we have developed a toolkit [1] for the verification of WSCL documents.

[1]http://keg.cs.tsinghua.edu.cn/persons/gzf/download/WSCL-toolkit-20061218-bin.tar.gz

The toolkit consists of four parts:

- A WSCL-to-promela translator written in Java
- A WSCL-to-dot translator written in Java
- Several shell scripts wrapping functions of the Java routines
- Several examples for demonstration

First we will introduce *wscl2pml.sh* and *genpml.sh*. The usage of these two scripts is shown as follows:

```
$ wscl2pml.sh <wscl-file>
             [promela-file]
$ genpml.sh <wscl-file> <property-file>
             [promela-file]
```

wscl2pml translates a WSCL document into a promela model by calling the java routine. *genpml* is an extended version of *wscl2pml*. Besides translating the WSCL document on command line by calling *wscl2pml*, it will append a property file to the output. The property file must have either .ltl or .never or .trace or .notrace as its extension. This script will take different actions according to the extension.

By executing these two scripts, we can get a promela file that can be fed into SPIN to be verified. It is highly recommended to use XSpin instead of running SPIN on the command line. XSpin is a GUI for SPIN written in Tcl/Tk. Not only can it hide many details about the command line options of SPIN, but also it can give a visualized view of the error trail if the property is violated.

Another shell script in our toolkit is *wscl2dot.sh*. This is a visualization tool for WSCL documents. Before using this tool, graphviz [2] should be installed first. *wscl2dot* can generate a description file in the format of the input language of graphviz, the DOT language. The usage of *wscl2dot* is the same as *wscl2pml*.

```
$ wscl2dot.sh <wscl-file> [dot-file]
```

It would be very helpful to obtain an overview of a WSCL document before analyzing it.

V. Conclusion and Future Work

In this paper, we present our work on the verification of conversations specified in WSCL. We study the verification tool SPIN, and try several methods to specify properties for WSCL documents. As a result, we have developed a toolkit to implement our ideas and methods.

We think the design of conversation protocol is essential to the future development of web services. With standardized conversation protocols, we can develop automation for code skeleton generation, runtime monitoring, transaction management and etc. As a result, the IT developers can concentrate more on the high level business logics. This will greatly reduce the development cycle and improve adaptability of IT systems.

We have also noticed the defects of WSCL. WSCL is a very simple specification. It specifies message sequence between two and only two participants. Paurobally has tried to extend it to support multi-parts conversation [2]. Additionally, WSCL is not an industry level specification, due to the lack of error handling, transaction management, etc. Many problems need to be solved before WSCL can be used in real-world applications. According to our work, we think WSCL may be extended to natively support properties; thus, properties can be published as a part of the conversation and be neutral to different model checkers.

[2]http://www.graphviz.org

Acknowledgment

This paper is supported by the IBM SUR project (Service Science and Technology).

References

[1] S. Vinoski, “Ws-nonexistent standards,” *Internet Computing, IEEE*, vol. 8, no. 6, p. 94, 2004, 1089-7801.

[2] S. Paurobally and N. R. Jennings, “Protocol engineering for web services conversations,” *Engineering Applications of Artificial Intelligence*, vol. 18, March 2005.

[3] R. J. Hall and A. Zisman, “Behavioral models as service descriptions,” in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA: ACM Press, 2004, pp. 163–172.

[4] T. Bultan, X. Fu, R. Hull, and J. Su, “Conversation specification a new approach to design and analysis of e-service composition.” in *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 403–410.

[5] X. Fu, T. Bultan, and J. W. Su, “Conversation protocols: A formalism for specification and verification of reactive electronic services,” in *Proceedings of Implementation And Application Of Automata*, ser. Lecture Notes In Computer Science. Springer-Verlag Berlin, 2003, vol. 2759, pp. 188–200.

[6] D. Beyer, A. Chakrabarti, and T. A. Henzinger, “Web service interfaces,” in *WWW '05: Proceedings of the 14th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2005, pp. 148–159.

[7] M. Y. Daniel and E. S. Robert, “Protocol specifications and component adaptors,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 292–333, 1997.

[8] U. Endriss, N. Maudet, F. Sadri, and F. Toni, “Protocol conformance for logic-based agents,” in *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*. Morgan Kaufmann Publishers, 2003.

[9] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti, “A priori conformance verification for guaranteeing interoperability in open environments,” in *Proceedings of 4th International Conference on Service Oriented Computing, Chicago, USA (4th–7th December 2006)*, ser. LNCS, vol. 4294, 2006, pp. 339–351. [Online]. Available: http://rewerse.net/publications/download/REWERSE-RP-2006-162.pdf

[10] Z. Xiangpeng, Y. Hongli, and Q. Zongyan, “Towards the formal model and verification of web service choreography description language,” in *Proc. of WS-FM 2006*, ser. LNCS 4184, Vienna, Austria, 2006. [Online]. Available: http://www.is.pku.edu.cn/~fmows/papers/cdl_verification.pdf

[11] W. van der Aalst, M. Dumas, A. ter Hofstede, N. Russell, H. Verbeek, and P. Wohed, “Life after bpel?” BPM Center Report BPM-05-23, BPMcenter.org, Tech. Rep., 2005. [Online]. Available: http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-23.pdf

[12] R. Kazhamiakin, M. Pistore, and M. Roveri, “Formal verification of requirements using spin: A case study on web services,” in *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 406–415.

[13] X. Fu, T. Bultan, and J. Su, “Wsat: A tool for formal analysis of web services,” in *LECTURE NOTES IN COMPUTER SCIENCE*. SPRINGER-VERLAG BERLIN, 2004, vol. 3114, pp. 510–514.

[14] A. Ankolekar, M. Paolucci, and K. Sycara, “Towards a formal verification of owl-s process models,” in *LNCS 3729*. Springer-Verlag Berlin Heidelberg, 2005, p. 37.

[15] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, Sep. 2003.

[16] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 411–420.