Ensuring Well-Formed Conversations between Control and Operational Behaviors of Web Services

Scott Bourne, Claudia Szabo, and Quan Z. Sheng

School of Computer Science
The University of Adelaide, SA 5005, Australia
{scott.bourne,claudia.szabo,michael.sheng}@adelaide.edu.au

Abstract. Despite a decade's active research and development, Web services still remain undependable. Designing effective approaches for highly dependable Web service provisioning has therefore become of paramount importance. Our previous work proposes a novel model that separates the service behavior into operational and control behaviors for flexible design, development, and verification of complex Web services. In this paper, we further this research with a set of conversation rules to facilitate the verification of rich conversations between control and operational behaviors. The rules are specified as temporal logic formulas to formally check rich conversation patterns. The proposed approach is realized using state-of-the-art technologies and experiments show its feasibility and benefits.

1 Introduction

Web services have been the focus of active research in the past decade [1–4]. Unfortunately, techniques on Web services design and deployment have not fully matured yet. Recent statistics show that only 28,600 Web services exist on the Web[1], and many have serious issues such as timeout, dependability and unexpected behavior, due to market pressures that require ad-hoc deployment without proper quality assurance. An important challenge remains verifying the soundness and completeness of a Web service at *design time*. This permits developers to identify major design flaws before costly development and will further the quality of the developed Web service [3, 5, 6].

Towards the verification of Web services at design time, our earlier work has proposed a novel model that separates the service behavior into *control* and *operational* behaviors, allowing for flexible design, development, and verification of complex Web services [3]. The control behavior guides the execution of the system and maintains a transactional state, while the operational behavior defines the underlying business logic. The conversation between control and operational behavior is formed by messages that direct the operational behavior and report events to the control behavior. The verification of the Web service can be

[1] `http://webservices.seekda.com`

thus translated into verifying that the control and operational behavior contain conversations that are well-formed.

In this paper, we define well-formed conversations between our proposed operational and control behaviors as conversations that start and end properly and for which several syntactic and semantic properties are met. This provides high flexibility and facilitates the definition and validation of rich conversations between operational and control behavior, at design time, before the Web service is developed. In particular, we propose to ensure well-formed conversations between control and operational behavior by defining a set of *conversation rules*, which ensure that (i) a conversation starts and ends correctly, and (ii) sequences of message exchange that may lead to deadlock and other undesired properties are not permitted. The main contributions of our work are as follows:

– A set of conversation rules to facilitate the verification of complex Web services.
– A service verification approach based on model checking that extracts temporal logic properties from a set of pre-defined rules.
– A prototype implementation that extends our existing set of tools with a conversation verifier to facilitate the automated verification of Web service design.

The remainder of this paper is organized as follows. Section 2 presents an overview of our Web service model for separating operational and control behaviors. Section 3 presents the conversation rules. We show how these rules are transformed to LTL properties in Section 4 and present an example in Section 5. Finally, we discuss related work and conlude in Section 6.

2 Background

In this section, we briefly overview our Web service behavior model presented in [3]. Our model enables a richer description of Web services by abstracting and separating a Web service's behavior into control and operational behaviors. The control behavior is an application-independent model of the state of the Web service from a transactional point of view, while the operational behavior represents the business logic that underpins the functionalities of the service. The execution of the operational behavior is guided by the control behavior, while events in the operational behavior influence the actions taken by the control behavior. Interested readers are referred to [3] for more details of this model.

We model the service behaviors using statecharts [7]. Figure 1(a) and (b) show the control and operational behavior models of WeatherWS, a weather information retrieval service. The operational behavior states are given meaningful names to reflect the underlying operations.

To enable inter-behavior communication, we propose a set of *message types*. These message types are classified as *initiation* messages and *outcome* messages. Initiation messages are sent from the control behavior for the purpose of directing the operational behavior, while outcome messages are replies that indicate the

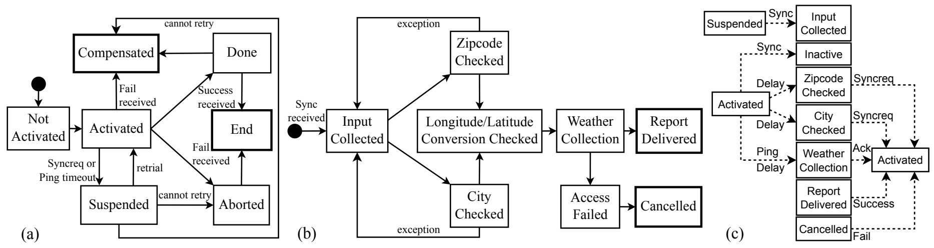


Fig. 1. Control (a) and operational behavior (b) of WeatherWS, with interactions (c).

current state of the operational behavior. The set of initiation messages include `Sync`, `Delay`, and `Ping`. `Sync` is used to trigger the execution of the operational behavior, `Delay` forces a response following an unacceptable delay, and `Ping` tests the liveness of an operational behavior state, triggering a timeout situation when no acknowledgement is received. Our outcome messages include `Success`, `Fail`, `Syncreq`, and `Ack`. `Success` and `Fail` indicate the commitment or abortion of the service. `Syncreq` requests another `Sync` to attempt forward recovery following an internal failure. `Ack` is used to respond to `Ping` and confirm the liveness of a state. The intra-behavior transition labels in Figure 1(c) shows the inter-behavior messages required for WeatherWS, while their effect is shown in (a) and (b).

However, guidance is still needed to produce a set of messages that ensure reliable and semantically correct conversations. We propose a set of rules to apply to behavior conversations to serve this purpose, as detailed in the following sections.

3 Conversation Rules

Our message types form conversations describing the execution of a Web service. However, there is a potential for invalid sequences, deadlocking situations, or incomplete sessions. We propose a list of conversation rules to ensure that behavior conversations are *well-formed*. We define well-formed conversations as sequences of messages that are correct, free of deadlock, and express the behavior of a service from invocation to termination. Our conversation rules specify correct sequences of message types, and ensure that a service is invoked and terminated correctly.

3.1 Conversation Sessions

Our proposed conversation rules apply to sequences of inter-behavior messages called *conversation sessions*. A conversation session is the ordered sequence of inter-behavior messages sent from the invocation of a service until both behaviors reach a termination state, i.e. the `End` or `Compensated` state in the control behavior.

A conversation session can be defined as a sequence of message types of length n. Each message is expressed as $m(t)$ where $m \in [Sync, Success, Fail, Syncreq, Delay, Ping, Ack]$ and t denotes the order such that $t \in [1, ..., n]$. For example:

Table 1. Conversation Rules

Name	Purpose	Conversation Rule
CR1	Initial Message	$\exists m \in [Sync], m(1)$
CR2	Final Messages	$\exists m \in [Success, Fail, Ping, Syncreq], m(n-1)$
CR3	Message Sequence	$\forall m_i \in [Sync, Ack]$, $\exists m_j \in [Success, Fail, Delay, Syncreq, Ping]$, $\forall t \in [0, ..., n-1], m_i(t) \Rightarrow m_j(t+1)$
CR4	Message Sequence	$\forall t \in [0, ..., n-1]$, $\exists m \in [Sync, Syncreq, Delay, Ping, Ack], m(t)$
CR5	Message Sequence	$\forall t \in [0, ..., n-2], Syncreq(t) \Rightarrow Sync(t+1)$
CR6	Message Sequence	$\exists m_j \in [Success, Fail, Syncreq]$, $\forall t \in [0, ..., n-1],\ Delay(t) \Rightarrow m_j(t+1)$
CR7	Message Sequence	$\exists m_j \in [Sync, Ack], \forall t \in [0, ..., n-1],\ Ping(t) \Rightarrow m_j(t+1)$
CR8	Message Sequence	$\forall m_i \in [Sync, Success, Fail, Ack, Delay, Syncreq]$, $\exists m_j \in [Sync, Success, Fail, Syncreq, Delay, Ping]$, $\forall t \in [0, ..., n-1], m_i(t) \Rightarrow m_j(t+1)$

– `Sync(1).Syncreq(2).Sync(3).Fail(4)` is a *well-formed* conversation session. It expresses a complete and deadlock-free execution of the service where each message logically follows the previous.

However, a lack of rules to ensure well-formed conversation sessions can lead to deadlocking states or incomplete sessions.

– `Sync(1).Syncreq(2).Delay(3)` is a deadlocking conversation session. The operational behavior is waiting for a `Sync` message before if can continue, while the control behavior is suspended until it receives a reply to `Delay`.
– `Sync(1).Ping(2).Ack(3).Ping(4).Ack(5)` is an incomplete conversation session, as the execution of the service has not fully terminated.

3.2 Conversation Rule Formalisms

We propose a set of conversation rules to enforce completeness of conversation sessions and logical message sequences, as shown in Table 1. The message sequence rules can be expressed as a series of `if-then` conditions as follows:

$$\forall m_i \in \mathcal{I}, \exists m_j \in \mathcal{J}, \forall t \in \mathcal{T}, m_i(t) \Rightarrow m_j(t+1)$$

where $\mathcal{I}, \mathcal{J} \subseteq [Sync, Success, Fail, Syncreq, Delay, Ping, Ack]$ and $\mathcal{T} \subseteq [1, ..., n-1]$. The set $\mathcal{I}$ identifies a set of message types, and $\mathcal{J}$ defines those that can immediately follow. This revises the formula presented in [3] by allowing rules to apply to several message types.

Rule CR1 specifies that all conversation sessions must begin with `Sync`, and CR2 defines the valid final messages for conversation sessions, ensuring that the control behavior does not enter a termination state before the operational behavior has completed. `Ping` can be the final message of a conversation session following an unrecoverable time-out, while `Syncreq` can be the final message when another `Sync` message cannot be sent (such as once a retrial limit has been exceeded).

Rule CR3 defines the set of valid messages to follow a `Sync` or `Ack` message. Once either of these message types are received, the operational behavior begins or resumes execution until completion or encountering a problem. This rule ensures that the control behavior does not send additional `Sync` messages while the operational behavior is executing.

Rule CR4 prevents the incorrect use of `Success` and `Fail` by specifying they cannot be used before the final message of the conversation session.

Rules CR5 and CR6 refer to messages that follow `Syncreq` and `Delay` messages respectively. A `Syncreq` message must only be replied with a `Sync` message (therefore it is impossible for `Syncreq` to be sent at $n-1$, as a session cannot end with `Sync`). Similarly, a `Delay` message must be immediately followed by a `Success`, `Fail` or `Syncreq` message.

Rule CR7 indicates that only a `Sync` or `Ack` message may follow a `Ping` message. We recall that when a `Ping` message is sent, either an `Ack` message is returned to confirm the liveness of an operational state, or a time-out situation occurs. In the case of a time-out, a `Sync` message can be sent to retry the process. This rule prevents sequences such as `Ping(t).Success(t+1)`, where the operational behavior has completed successfully, but the control behavior is still waiting for an acknowledgement and cannot proceed. Rule CR8 is also needed to ensure that `Ack` can only follow `Ping`.

The conversation rules also imply other desirable properties, such as preventing the same message type to be sent consecutively, and ensuring every `Sync` message eventually receives an outcome message (`Success`, `Fail` or `Syncreq`), excepting a time-out. By defining initial messages, final messages, and valid message sequences, our proposed rules set can ensure complete and correct conversation sessions.

4 From Conversation Rules to Temporal Logic

To formally verify a service design against our conversation rules, we explore the use of model checking [8] to ensure conformance to pre-defined temporal properties describing our proposed rules. We use Linear Temporal Logic (LTL) [9] for this purpose. LTL expresses properties of a system model over a linear and discrete timeline by using *temporal operators* over model variables.

While our conversation rules can be applied to a simple sequence of message types, the LTL properties must apply to a complete service model. This poses two challenges when producing LTL transformations. Firstly, the complexity of the service model can cause state delays between certain inter-behavior messages. Secondly, there is a need to extract the state of the conversation session from the service model. To address these issues, the LTL properties consider potential message delays where appropriate and use a set of proposed conversation variables.

Table 2. Conditions for Message Processing

Message	**Processed Condition**
`Sync`	The operational behavior begins or resumes execution.
`Success, Fail`	The control state transitions from the Activated state.
`Syncreq`	A `Sync` message is sent in reply or a termination state is entered.
`Delay`	A `Success`, `Fail` or `Delay` message is sent in reply.
`Ping`	An Ack message is sent in reply.
`Ack`	Automatically processed in the following state.

Table 3. LTL Transformations of Conversation Rules

CR1	$(IM = nil \wedge OM = nil) \cup (IM = Sync \wedge IP = FALSE \wedge OM = nil)$
CR2	$\square((((IM = Sync \vee IM = Delay) \wedge IP = FALSE) \vee (OM = Ack \wedge OP = FALSE)) \rightarrow \diamond((OM \neq Ack \wedge OP = FALSE) \vee (IM \neq Sync \wedge IM \neq Delay \wedge IP = FALSE)))$
CR3	$\square(((IM = Sync \wedge IP = FALSE) \vee (OM = Ack \wedge OP = FALSE)) \rightarrow \bigcirc ((IP = TRUE \wedge OP = TRUE) \vee ((IP = FALSE \wedge (IM = Ping \vee IM = Delay)) \vee (OP = FALSE \wedge (OM = Success \vee OM = Fail \vee OM = Syncreq)))))$
CR4	$\square((OM = Success) \rightarrow \bigcirc (OM = Success \wedge OP = TRUE \wedge IP = TRUE))$ $\square((OM = Fail) \rightarrow \bigcirc (OM = Fail \wedge OP = TRUE \wedge IP = TRUE))$
CR5	$\square((OM = Syncreq \rightarrow \bigcirc(OM = Syncreq \wedge OP = TRUE \wedge IP = TRUE)) \vee ((OM = Syncreq \wedge OP = FALSE) \rightarrow \bigcirc((OM = Syncreq \wedge OP = FALSE) \vee (OP = TRUE \wedge IM = Sync \wedge IP = FALSE))))$
CR6	$\square((IM = Delay \wedge IP = FALSE) \rightarrow \bigcirc((IP = Delay \wedge IP = FALSE \wedge OP = TRUE) \vee (IP = TRUE \wedge OP = FALSE \wedge (OM = Fail \vee OM = Success \vee OM = Syncreq))))$
CR7	$\square((IM = Ping \wedge IP = FALSE) \rightarrow \bigcirc ((IP = FALSE \wedge OP = TRUE \wedge (IM = Sync \vee IM = Ping)) \vee (OM = Ack \wedge OP = FALSE \wedge IP = TRUE)))$
CR8	$\square((OM = Ack \wedge OP = FALSE) \rightarrow (IM = Ping \wedge IP = TRUE))$ $\square((OP = FALSE) \rightarrow \bigcirc\neg(OM = Ack \wedge OP = FALSE))$

The LTL transformations of our conversation rules utilize temporal operators over four proposed variables to express the conversational state. Initiation and outcome message are denoted by the variables `IM` and `OM` respectively. Both variables are initialized at `nil`. We also propose two boolean variables, `IP` and `OP` to indicate when their corresponding message is active or processed. Their values are set to `FALSE` upon sending and `TRUE` after processing. The processing conditions for each message type is shown in Table 2. We employ the following operators over these conversation variables: $\bigcirc$ for the next state, $\diamond$ for at least one future state, $\square$ for all states, and $\cup$ for one property to hold until another is met.

Table 3 shows the LTL transformations of the conversation rules. The transformations of some rules, CR1 and CR4, are straightforward. We enforce CR2 by ensuring at least one valid final message always follows a non-final message.

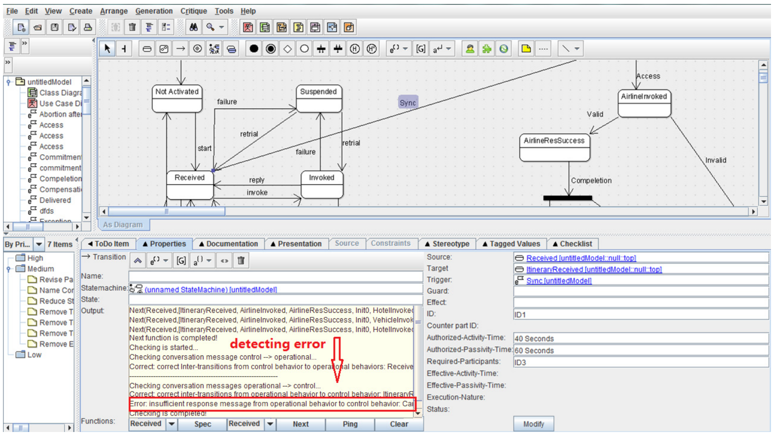
Delays of several states can potentially occur following `Sync`, `Ping`, `Syncreq`, and `Ack`. Therefore, the transformations of rules CR3, CR5, CR6 and CR7 allow unchanged conversational states as one valid *next* state. Rule CR8 requires two LTL properties to express; one to ensure that `Ping` is the initiation message to precede `Ack`, and another to prevent outcome messages between `Ping` and `Ack`.

These LTL properties can be used to verify a complex service designed as control and operational behaviors, by ensuring it only produces well-formed conversation sessions. A service design can be checked against the rules by creating a model that contains our message variables and using a model checking tool to verify that none of the rules are violated.

5 System Implementation and Validation

We implemented the approach proposed in this paper by extending our existing prototype system for the design and verification of Web services. Our system is implemented in Java and uses state-of the art technologies such as XML, SOAP, WSDL, and model checking. Users can access the system via the user interface shown in Figure 2, to compose, verify and execute complex services as interacting control and operational behaviors. The NuSMV model checker [10] is used to verify service designs within this system. The prototype enables a service design to be transformed into the input language of NuSMV, and then verified for conformance to the LTL transformations of our conversation rules. The prototype is an extension of the system in [3].

To evaluate our proposed approach, we conducted experiments using the implemented prototype and the WeatherWS example shown in Figure 1. The control behavior, operational behavior, and inter-behavior conversations were modeled in SMV, the input language of NuSMV. We applied the NuSMV model



Name:
Statemachine: (unnamed StateMachine) [untitledModel]
State:
Output:

```
Next(Received,[ItineraryReceived, AirlineInvoked, AirlineResSuccess, Init0, HotelInvoke
Next(Received,[ItineraryReceived, AirlineInvoked, AirlineResSuccess, Init0, VehicleInvok
Next(Received,[ItineraryReceived, AirlineInvoked, AirlineResSuccess, Init0, HotelInvoke
Next function is completed!
Checking is started...
Checking conversation message control --> operational...
Correct: correct Inter-transitions from control behavior to operational behaviors: Receive
------------------------------------------------
Checking conversation messages operational --> control...
Correct: correct inter-transitions from operational behavior to control behavior: ItineraryR
Error: insufficient response message from operational behavior to control behavior: Ca
Checking is completed!
```

Source: Received [untitledModel::null::top]
Target: ItineraryReceived [untitledModel::null::top]
Trigger: Sync [untitledModel]
Guard:
Effect:
ID: ID1
Counter part ID:
Authorized-Activity-Time: 40 Seconds
Authorized-Passivity-Time: 60 Seconds
Required-Participants: ID3
Effective-Activity-Time:
Effective-Passivity-Time:
Execution-Nature:
Status:

Fig. 2. Specifying Service Behaviors

checker to verify that the conversational behavior of this model does not violate any of the LTL properties produced in Section 4. We verified ten testcases with artificially introduced errors. If any of the properties are violated by the model, NuSMV produces a state sequence that leads to the contradiction. Our SMV transformation of WeatherWS as it appears in Figure 1 satisfied the set of conversation rules.

6 Discussion and Conclusion

The verification of Web service behavior remains an important challenge despite active research and development over the last decade. Our work facilitates the flexible verification of Web services at design time by modeling Web service behavior as a conversation between an operational behavior that defines the underlying business logic of the system, and an application-independent control behavior that guides the execution of the operational behavior. We propose a set of conversation rules that are specified as temporal properties and verify using a model checker.

Most existing work on Web service conversation modeling has focused on the interactions between a deployed service and a client. Technical specifications have been proposed to express the conversational requirements of a complex service as part of an interface [4, 11]. In contrast, we apply temporal logic and model checking to ensure that all possible conversations follow a set of rules for correctness and completeness. In a similar effort, Kova et al. [12] study the mapping between the control and operational behaviors and also propose to verify the conversations using LTL properties and the NuSMV model checker. In their approach, the behaviors are merged into a single model that express the possible flow of control states. Model checking is used to verify that the transitions between operational states do not violate transitions defined in the control behavior model. Our work differs by defining message types and rules for the communication between the two behavior models. By using messages that can dictate the transitions between states in both models, we are able to model a wider range of transactional behavior (such as pinging operations and responding to delays).

Future work includes expanding the applicability of the control and operational behaviors by including handling for more sophisticated workflow patterns such as iteration and parallel execution. We will also consider to include context information in conversation rules, such as considering the temporal properties of failed operations when attempting corrective action via the control behavior.

References

1. Yu, Q., Liu, X., Bouguettaya, A., Medjahed, B.: Deploying and Managing Web Services: Issues, Solutions, and Directions. The VLDB Journal 17(3), 537–572 (2008)
2. Vieria, M., Laranjeiro, N., Madeira, H.: Benchmarking the Robustness of Web Services. In: Proceedings of the 13th International Symposium on Pacific Rim Dependable Computing (2007)

3. Sheng, Q., Maamar, Z., Yahyaoui, H., Bentahar, J., Boukadi, K.: Separating Operational and Control Behaviors: A New Approach to Web Services Modeling. IEEE Internet Computing 14(3), 68–76 (2010)
4. Benatallah, B., Casati, F., Toumani, F.: Web Service Conversation Modeling: A Cornerstone for E-Business Automation. IEEE Internet Computing 8(1) (2004)
5. Bhiri, S., Perrin, O., Godart, C.: Ensuring Required Failure Atomicity of Composite Web Services. In: Proceedings of the 14th International World Wide Web Conference, pp. 138–147. ACM (2005)
6. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A Framework for Fault-tolerant Composition of Transactional Web Services. IEEE Transactions on Services Computing 3(1), 46–59 (2010)
7. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (1996)
8. Clarke, E.M.: Model Checking. In: Ramesh, S., Sivakumar, G. (eds.) FST TCS 1997. LNCS, vol. 1346, pp. 54–56. Springer, Heidelberg (1997)
9. Emerson, E.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, vol. 2, pp. 995–1072 (1990)
10. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
11. Ardissono, L., Goy, A., Petrone, G.: Enabling Conversations with Web Services. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 819–826. ACM (2003)
12. Kova, M., Bentahar, J., Maamar, Z., Yahyaoui, H.: A Formal Verification Approach of Conversations in Composite Web Services using NuSMV. In: Proceedings of the Conference on New Trends in Software Methodologies, Tools and Techniques, pp. 245–261. IOS Press (2009)