Realizability of Conversation Protocols With Message Contents

Xiang Fu Tevfik Bultan Jianwen Su
Department of Computer Science,
University of California, Santa Barbara, CA 93106
{fuxiang,bultan,su}@cs.ucsb.edu

Abstract

A conversation protocol is a top-down specification framework which specifies desired global behaviors of a web service composition. In our earlier work [6] we studied the problem of realizability*, i.e., given a conversation protocol, can a web service composition be synthesized to generate behaviors as specified by the protocol. Several sufficient realizability conditions were proposed in [6] to ensure realizability. Conversation protocols studied in [6], however, are essentially abstract control flows without data semantics. This paper extends the work in [6] and achieves more accurate analysis by considering data semantics. To overcome the state-space explosion caused by the data content, we propose a symbolic analysis technique for each realizability condition. In addition, we show that the analysis of the autonomy condition can be done using an iterative refinement approach.*

1. Introduction

Recently, verification (especially model checking) of web service compositions has been attracting much attention [4, 5, 12]. Formal verification techniques can provide a way to develop highly dependable web services: preset system goals can be guaranteed to be satisfied by the implementations prior to the service deployment. However, before the application of the verification techniques, the first challenge is to establish a formal modeling and specification framework for web service compositions. There are numerous competing standards for composition of web services (e.g. BPEL4WS [1], WSCI [15], DAML-S [3]), which complicates the task of formal modeling and specification. Additionally, certain characteristics of web services cause important problems. For example, asynchronous communication (supported by messaging platforms such as Java Message service (JMS) [9] and Microsoft Message Queuing service (MSMQ) [11]) causes LTL model checking to be undecidable [6].

In our previous work [2, 6], we established a conversation oriented framework to specify web service compositions and reason about their global behaviors. Each participant (a peer) of a composition is characterized using a finite state automaton (FSA), with the set of input/output message classes (without message contents) as the FSA alphabet. To simulate asynchronous communication, each peer is equipped with a FIFO queue to store incoming messages. The behaviors generated by a composition of peers can be characterized using the set of message sequences (*conversations*) exchanged among peers. Clearly Linear Temporal Logic (LTL) can be easily extended to this conversation based framework [6] and specify desired system goals like "when a request of `cancel` arrives, eventually the server should respond with a `cancel confirmation`".

In general, there are two different ways to specify a web service composition: 1) The bottom-up approach, which is favored by most industry standards such as WSDL [13] and BPEL4WS [1], where each participant of the composition is specified first and then the composed system is studied; and 2) the top-down approach (e.g., conversation policies [8] and Message Sequence Charts [10, 4]) where the set of desired message events is specified and detailed peer implementations are left blank. In [2] we generalized the notion of conversation policy from two peers to arbitrary number of peers, and proposed the notion of *conversation protocol*. Compared with the bottom-up approach, the expressive power of a conversation protocol is weaker, however this leads to certain benefits in verification [2].

A conversation protocol, however, is not always *realizable*. There may not exist any peer implementations whose composition generates exactly the same set of conversations as specified by the protocol. In [6] we proposed three sufficient *realizability conditions* to restrict control flows of a conversation protocol, and when these conditions are satisfied, the projections of the protocol to each peer are guaranteed to realize the protocol. One byproduct of the realizability analysis is avoiding the undecidability of verification problems caused by the asynchronous communication. Our realizability analysis leads to a 3-step specification and verification strategy: 1) A web service composition is specified using a realizable conversation protocol. 2) Desired LTL properties are verified on the conversation protocol. 3) The conversation protocol is projected to each peer, and the composition of these peer implementations will preserve the LTL properties that are previously verified.

Contributions of the present paper: While the framework presented in [2, 6] introduces the concept of conversation protocol and realizability conditions, it is still a step away from practical web service applications since the message contents and the data semantics are ignored. It is interesting to raise the following question: can the realizability analysis in [6] work when data semantics is associated with a conversation protocol? Or, to be more precise, given a *guarded* conversation protocol where each transition in the protocol is equipped with a *guard* to manipulate data, let *skeleton* be the standard guardless conversation protocol generated from the guarded protocol by removing all data and guards. If the skeleton is realizable, does it imply that the guarded conversation protocol is realizable? This paper answers the above questions, and also provides symbolic analysis techniques for the realizability conditions.

This paper is organized as follows. Section 2 lays out the formal definition of conversation based framework for web service compositions. Section 3 introduces a light-weight skeleton analysis for guarded conversation protocols. Section 4 presents the error-trace guided refined analysis for the autonomy condition. Section 5 briefly discusses the symbolic analysis for the other two realizability conditions. Section 6 concludes the paper.

2. A Formal Specification Framework

This section presents a conversation oriented framework to specify web service compositions. As we mentioned earlier, a web service composition can be described in a bottom-up or top-down fashion. Both specification approaches are based on a notion called "Guarded Finite State Automata" (GFSA), which incorporates data semantics into the framework. In this section, we begin with the introduction of composition schema, the interconnection pattern of a web service composition. Then we introduce GFSA, based on which we present both the top-down and bottom-up specification approaches. Finally we define the notion of realizability.

2.1. Composition Schema

A *composition schema* is a tuple (P, M, Σ) where P is a finite set of peers, M is a finite set of message classes, and Σ is a (finite or infinite) set of of messages. Generally a composition schema defines the composition infrastructure of a set of web services. Each message class $c \in M$ is transmitted on one peer to peer channel, and each message class has a finite set of attributes where each attribute has a static data type such as integer, boolean, enumerated, or character. Each message $m \in \Sigma$ is an *instance* of some message class in M.

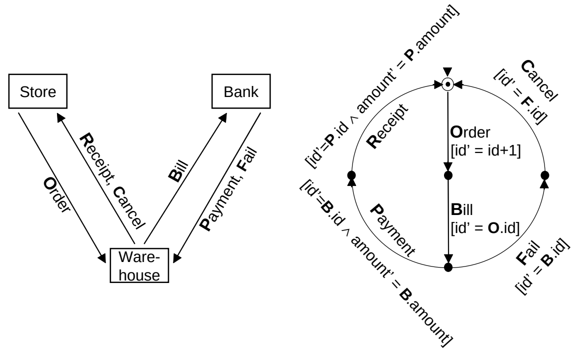


Figure 1. An Online-Store Example

For example, the diagram on the left side of Figure 1 defines a composition schema for an online-store web service composition. There are three peers, *Store*, *Bank* and *Warehouse*. Message classes, such as `Order`, `Bill`, and `Payment`, are transmitted among these three peers. In the rest of this paper, we assume that message classes `Bill`, `Payment` and `Receipt` have two integer attributes `id` and `amount`, and the rest of message classes in Figure 1 has one attribute `id` only. As a message is an instance of a message class, it is written in the form of "`class`(`contents`)". For example, **B**$(100, 2000)$ stands for a `Bill` whose `id` is 100 and `amount` is 2000. Here `Bill` is represented using its capitalized first letter **B**. Finally notice that in this paper message contents are "flattened". A version of our model with more complex type support such as XML Schema [16] is presented in [5].

2.2. Top-down approach: Conversation Protocol

A *(guarded) conversation protocol* is a tuple $\langle (P, M, \Sigma), \mathcal{A} \rangle$, where (P, M, Σ) is the composition schema, and $\mathcal{A}$ is a Guarded Finite State Automaton (GFSA). Figure 1 shows an example guarded conversation protocol where the diagram on the right side is its GFSA specification.

A GFSA is a tuple $(M, \Sigma, T, s, F, \Delta)$, where M and Σ are the set of message classes and messages respectively, T is a finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states, and Δ is the transition relation. Each transition $\tau \in \Delta$ is of the form $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of the transition τ,

$c \in M$ is a message class and g is the *guard* of the transition. The guard of a transition expresses the relationships between the message that is being sent and the last message of each class that is sent or received by the sender of the message being sent. For example, the guard of the transition to send Order is: $\texttt{Order.id}' = \texttt{Order.id}+1$, which intends to increment the value of attribute id by 1 whenever a new Order message is sent. In a guard, a *primed* message attribute stands for its "next value" after execution of the transition, and a *non-primed* attribute stands for the value of this attribute in the "latest" message of that message class. (If for a message class there is no instance received or sent yet, then attribute values for that message class are initialized nondeterministically.)

Example 2.1 The guarded conversation protocol in Figure 1 describes the following scenario: an order is placed by *Store* to *Warehouse*, and then *Warehouse* sends a Bill to the *Bank*. The *Bank* either responds with the Payment or rejects with a Fail message. Finally *Warehouse* issues a Receipt or a Cancel message. The guards determine the contents of the messages. For example, the id and amount of each Payment must match those of the latest Bill message. ▮

It is straightforward to define the automata configuration and the derivation relation between configurations for a GFSA. Based on the derivation relation, the notion of a run and accepted words can be defined. Details can be found in the complete version of this paper [7]. For example, one possible word accepted by the GFSA in Figure 1 is:

$$\mathbf{O}(0), \mathbf{B}(0,100), \mathbf{P}(0,100), \mathbf{R}(0,100), \mathbf{O}(1), \mathbf{B}(1,200), \mathbf{F}(1), \mathbf{C}(1)$$

2.3. Bottom-up Service Composition

Now we introduce the traditional bottom-up specification of the composition of a set of web services. A *web service composition* is a tuple $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where (P, M, Σ) is the composition schema, and $\mathcal{A}_i$ is the peer implementation for peer p_i, where $i \in [1..n]$ and $n = |P|$. Each $\mathcal{A}_i$ is a GFSA that is slightly different than the GFSA used to specify a conversation protocol. In $\mathcal{A}_i$ there are two types of transitions: 1) send-transition, which sends out a message and has a guard to determine the contents of the message being sent, and 2) receive-transition, which consumes a message from the input queue. Send and receive-transitions are denoted using "!" and "?", respectively. Figure 2 shows an example web service composition which "implements" the conversation protocol in Figure 1.

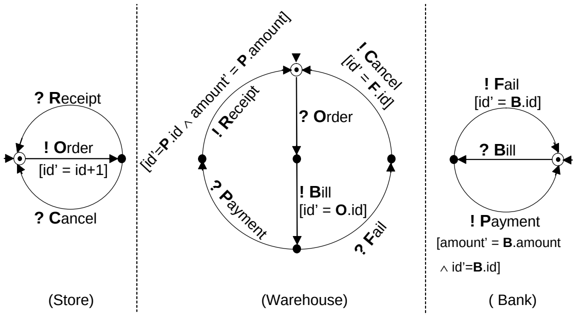


Figure 2. A Realization of Figure 1

Next we present the notion of "conversation" to capture the global behaviors that are generated by a web service composition. Given a web service composition $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where $n = |P|$ and $k = |M|$, a *global configuration* of S is a $(2n+3)$-tuple of the form

$$(Q_1, t_1, ..., Q_n, t_n, w, \vec{s}, \vec{c})$$

where for each $j \in [1..n]$, $Q_j \in (\Sigma^i_j)^*$ is the queue content of peer p_j (where Σ^i_j is the set of input messages of peer p_j), t_j is the state of p_j, $w \in \Sigma^*$ is the global watcher which records the sequence of messages that have been transmitted, and message vectors $\vec{s}, \vec{c}$ record the latest *sent* and *consumed* instances (resp.) for each message class. Basically a global configuration can be regarded as a "snapshot" of the system state of a web service composition. We can naturally define a derivation relation between two configurations γ and γ', written as $\gamma \rightarrow \gamma'$, if from γ some peer sends out a message, or consume a message, and the whole system advances into the configuration γ'. (Detail definition can be found in [7].) We say a word $w \in \Sigma^*$ is a *conversation* of $\mathcal{S}$ if we can find a sequence of configurations $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_j$ such that γ_0 is the initial configuration, and γ_j is a "final configuration" where each peer is in a final state and all input queues are empty. Let $\mathcal{C}(\mathcal{S})$ denote the set of conversations of a web service composition $\mathcal{S}$. We say that $\mathcal{S}$ *realizes* a conversation protocol $\mathcal{P}$ if $\mathcal{C}(\mathcal{S}) = L(\mathcal{P})$. A conversation protocol $\mathcal{P}$ is *realizable* if there is a web service composition that realizes $\mathcal{P}$.

3. Skeleton Analysis

Our previous work [6] introduced a realizability analysis for guardless conversation protocols where data semantics are abstracted away. This section investigates the following question: Can the results in [6] be applied to a guarded conversation protocol by analyzing its "skeleton"? We first briefly introduce the realizability conditions defined in [6], and then present the skeleton analysis.

3.1. Previous Work

We start with three motivating examples (non-realizable guardless conversation protocols) for realizability conditions. The protocol at Figure 3(a) specifies a single conversation $\alpha\beta$, and the subscript of each message class describes its sender and receiver, e.g., α is from peer A to B. Obviously Figure 3(a) is not realizable, because any implementation of peers A, B, C, D which generates $\alpha\beta$ can generate $\beta\alpha$ as well. Similarly Figure 3(b) is not realizable since it does not allow the conversation $\gamma\alpha$. Figure 3(c) is more interesting. According to the protocol, peer A and B can take the left and right branch of Figure 3(c) respectively, without noticing the other peer is taking a different branch. Here A first sends out the message α, and α is stored in the queue of B; then B sends out β, and β is stored in the queue of A. Peer A and B consume the messages in their queues, and finally B sends out message γ. Hence a conversation $\alpha\beta\gamma$ is produced by the peer implementations, however, it is not allowed by the conversation protocol shown in the Figure 3(c).

In [6], we proposed three sufficient *realizability conditions* for guardless conversation protocols. A *guardless protocol* is a tuple $\langle (P, M), \mathcal{A} \rangle$ where messages do not have any content (i.e., each message is completely specified by its message class), and $\mathcal{A}$ is a standard FSA. In [6], we showed that when the three conditions are satisfied, the projections of a guardless protocol to each peer are guaranteed to realize the protocol. In order to describe the realizability conditions, we need to define the projection and join operations. Given a message alphabet $\Sigma = \Sigma_1 \cup ... \cup \Sigma_n$ which is the union of pairwise disjoint message sets $\Sigma_1, ..., \Sigma_n$, for any word $w \in \Sigma^*$, its *projection* to Σ_i, written as $\pi_i(w)$, is a word in Σ_i^*. $\pi_i(w)$ is generated from w by deleting all messages that do not belong to Σ_i. Given n languages $L_1, ..., L_n$ where for each $i \in [1..n] : L_i \subseteq \Sigma_i^*$, the *join* of $L_1, ..., L_n$ is defined as follows:

$$\bowtie (L_1, ..., L_n) = \{w \mid \forall i \in [1..n] : \pi_i(w) \in L_i\}.$$

For a language $L \subseteq \Sigma^*$, its *join closure* is defined as: $\textsc{joinc}(L) = \bowtie (\pi_1(L), ..., \pi_n(L))$.

1) Lossless join: A guardless conversation protocol $\langle (P, M), \mathcal{A} \rangle$ is *lossless join* if the join closure of $\mathcal{A}$ is equivalent to $\mathcal{A}$, i.e., $\textsc{joinc}(L(\mathcal{A})) = L(\mathcal{A})$. We can check this property by projecting $\mathcal{A}$ to each peer, and then constructing the Cartesian product of all the projections, and then checking the equivalence between the Cartesian product and $\mathcal{A}$.

2) Synchronous compatibility: Formally, a conversation protocol $\langle (P, M), \mathcal{A} \rangle$ is *synchronous compatible* if the following formula holds:

$$\forall w \in L^*(\mathcal{A}),\ \forall i, j \in [1..|P|],\ \forall m \in M_i^{out} \cap M_j^{in} :$$
$$\pi_i(wm) \in \pi_i(L^*(\mathcal{A})) \ \Rightarrow\ \pi_j(wm) \in \pi_j(L^*(\mathcal{A}))$$

Here $L^*(\mathcal{A})$ is the prefix language of $\mathcal{A}$, i.e., $L^*(\mathcal{A}) = \{w \mid w \text{ is a prefix of } w', \text{ and } w' \in L(\mathcal{A})\}$. Given a guardless conversation protocol $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, to check if it is synchronous compatible, we construct the Cartesian product of the *determinized* projections of $\mathcal{A}$ to peers. If we cannot find a state in the Cartesian product such that a peer p_i is ready to send a message to peer p_j but peer p_j is not ready to receive the message, $\mathcal{P}$ is synchronous compatible.

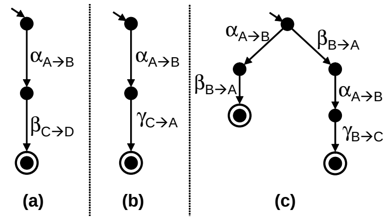


Figure 3. Non-realizable Examples

3) Autonomy: A guardless conversation protocol is *autonomous* if for each peer p_i and each finite prefix w of a conversation, at most one of the following three conditions hold: a) the next transitions of p_i (including transitions that are reachable through ϵ-transitions) are all send operations, b) the next transitions of p_i (including transitions that are reachable through ϵ-transitions) are all receive operations, or c) p_i is in a final state.

In [6] we showed that a guardless conversation protocol is realizable if the above three conditions are satisfied. It is not hard to see that the conversation protocols at Figure 3(a), Figure 3(b), and Figure 3(c) violate the lossless join, synchronous compatibility, and autonomy respectively (while satisfying the other two conditions).

3.2. A Fourth Condition

Given a GFSA $\mathcal{A} = (M, \Sigma, T, s, F, \Delta)$, its skeleton, denoted as $\text{skeleton}(\mathcal{A})$, is a standard FSA (M, T, s, F, Δ') where Δ' is obtained from Δ by replacing each transition $(s, (c, g), t)$ with (s, c, t). Note that $L(\text{skeleton}(\mathcal{A})) \subseteq M^*$, while $L(\mathcal{A})$ is a subset of Σ^*. For a guarded conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, we can always construct a guardless conversation protocol $\langle (P, M), \text{skeleton}(\mathcal{A}) \rangle$ as defined in [2, 6]. We call this protocol the *skeleton protocol* of $\mathcal{P}$.

Now, one natural conjecture is: If the skeleton protocol of a guarded conversation protocol is realizable, does this imply that the guarded protocol is realizable? In Figure 4 we give a counter example.

Example 3.1 The guarded conversation protocol shown in Figure 4 has four peers A, B, C, D. There are two message classes in the system: α is from A to B and β is from C to D. Both message classes have an attribute a. The protocol specifies two possible conversations $\alpha(1)\beta(1)$, and $\beta(2)\alpha(2)$. The skeleton of this protocol is realizable, however the protocol itself is not, because any implementation

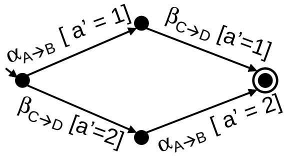


Figure 4. A Counter Example

that generates the specified conversations will also generate the conversation $\beta(1)\alpha(1)$ as well. ▮

Here we propose an extra condition that restricts the guards of a guarded conversation protocol.

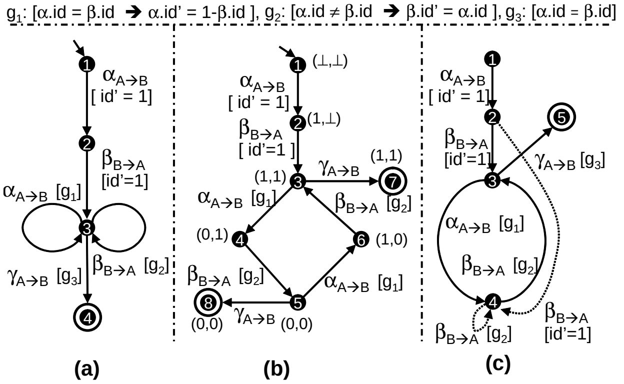
4) Deterministic Guards: To check the deterministic guard condition, for each peer p_i, we analyze $\pi_i(\mathcal{A})$, the projection of $\mathcal{A}$ to peer p_i. We determinize $\pi_i(\mathcal{A})$ as a guardless standard FSA, ignoring the guards. Each state in this determinized automaton corresponds to a set of states in $\pi_i(\mathcal{A})$. For each state in the determinized automaton we collect all the transitions which start from the the corresponding states in $\pi_i(\mathcal{A})$ and are labeled with the same message class. We require that at each state for all send transitions at that state there can be at most one guard for each message class, i.e., if there are two send transitions with the same message class, their guards have to be identical. A guarded conversation protocol satisfies the deterministic guards condition, if such a check succeeds for each peer. It is not hard to see that Figure 4 violates the deterministic guards condition, because peer A has two different guards when sending out α.

Theorem 3.2 A guarded conversation protocol is realizable if it satisfies the deterministic guards condition, and its skeleton protocol satisfies the lossless join, synchronous compatibility and autonomy condition.

Based on the above theorem, we have a light-weight realizability analysis for guarded conversation protocols. We check the first three realizability conditions on the skeleton of a conversation protocol (i.e, without considering the guards), and then examine the fourth realizability condition by syntactically check the guards (but actually without analyzing their data semantics).

4. Refined Analysis of Autonomy

Skeleton analysis may not be very precise. In Figure 5 we show a realizable conversation protocol where skeleton analysis fails. The alternating-bit protocol shown in Figure 5(a) consists of two peers A and B. Message class α is request, and message class β is acknowledgment. Both message classes contain an "`id`" attribute. Message class γ is the end-conversation notification. The protocol states that the `id` attribute of requests from A alternates between 0 and 1, and every acknowledgment β must match the `id`. Obviously, when the protocol is projected to each peer, we still get the same GFSA (except "!" and "?" are added for output/input transitions).



(a) (b) (c)

Figure 5. Alternating Bit Protocol

Let $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ be the three conversation protocols shown in Figure 5. It is not hard to see that, the projection of $\mathrm{skeleton}(\mathcal{A}_a)$ to peer A does not satisfy the autonomy condition, because at state 3, there are both input and output transitions. However, $\mathcal{A}_a$ is actually autonomous. If we explore each configuration of $\mathcal{A}_a$, we get $\mathcal{A}_b$, the "equivalent" conversation protocol of $\mathcal{A}_a$. The pair of values associated with each state in $\mathcal{A}_b$ stands for the `id` attribute of α and β. It is obvious that $\mathcal{A}_b$ satisfies the autonomy condition, and hence $\mathcal{A}_a$ should satisfy autonomy as well. In fact to prove that $\mathcal{A}_a$ is autonomous we do not even have to explore each of its configurations like $\mathcal{A}_b$. As we will show later, it suffices to show $\mathcal{A}_c$ is autonomous. Finally notice that $L(\mathcal{A}_a) = L(\mathcal{A}_b) = L(\mathcal{A}_c)$.

The examples in Figure 5 motivates a refined analysis: given a conversation protocol $\mathcal{A}$, we can first check its skeleton. If the skeleton analysis fails, we can refine the protocol (e.g. refine $\mathcal{A}_a$ and get $\mathcal{A}_c$), and apply the skeleton analysis on the refined protocol. We can repeat this procedure until we reach the most refined protocol which actually plots the transition graph of the configurations of the original protocol (such as $\mathcal{A}_b$ to $\mathcal{A}_a$). Our refined analysis of autonomy is based on the notion of *simulation*, which is defined as below.

A transition system is a tuple (M, T, s, Δ) where M is the set of labels, T is the set of states, s the initial state, and Δ the transition relation. Generally a transition system can be regarded as an FSA without final states. On the other hand, a standard FSA (M, T, s, F, Δ) can be regarded as a transition system of (M, T, s, Δ); and a GFSA $(M, \Sigma, T, s, F, \Delta)$ can be regarded as a transition system of the form $(\Sigma, T', s', \Delta')$ where T' contains all configurations of the GFSA, and Δ' defines the derivation relation between configurations.

Definition 4.1 A transition system $\mathcal{A}' = (M', T', s', \Delta')$

is said to *simulate* another $\mathcal{A} = (M, T, s, \Delta)$, written as $\mathcal{A} \preceq \mathcal{A}'$, if there exists a mapping $\rho : T \rightarrow T'$ and $\varrho : M \rightarrow M'$ such that for each (s, m, t) in Δ there is a $(\rho(s), \varrho(m), \rho(t))$ in Δ'. Two transition systems $\mathcal{A}$ and $\mathcal{A}'$ are said to be *equivalent*, written as $\mathcal{A} \simeq \mathcal{A}'$, if $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{A}' \preceq \mathcal{A}$. ▮

Example 4.2 For the three conversation protocols $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ in Figure 5, the following is true:

$$\text{skeleton}(\mathcal{A}_b) \preceq \text{skeleton}(\mathcal{A}_c) \preceq \text{skeleton}(\mathcal{A}_a)$$

For example, for $\text{skeleton}(\mathcal{A}_c) \preceq \text{skeleton}(\mathcal{A}_a)$, the mapping ρ maps states $1, 2, 3, 4, 5$ in $\text{skeleton}(\mathcal{A}_c)$ to states $1, 2, 3, 3, 4$ of $\text{skeleton}(\mathcal{A}_a)$ respectively, and the mapping ϱ is the identity function which maps each message class to itself.

For another example, $\mathcal{A}_a \preceq \text{skeleton}(\mathcal{A}_a)$, and similar relation holds for any GFSA and its skeleton. We can construct the mappings for a GFSA to its skeleton. Since a configuration of GFSA is of the form $(t, \vec{m})$ where t records the local state and $\vec{m}$ is a vector of message instances for each message class, for each $(t, \vec{m})$ mapping ρ maps it to t in the skeleton. For each message m, mapping ϱ maps to its class. Finally it is not hard to see that $\mathcal{A}_a \simeq \mathcal{A}_b \simeq \mathcal{A}_c$. ▮

Intuitively when $\mathcal{A} \preceq \mathcal{A}'$, each word accepted by $\mathcal{A}$ will find a corresponding word accepted by $\mathcal{A}'$, and $\mathcal{A}'$ can contain “more” words than $\mathcal{A}$. It is not hard to infer the following lemma.

Lemma 4.3 For any GFSA $\mathcal{A}$, $\mathcal{A} \preceq \text{skeleton}(\mathcal{A})$.

Lemma 4.4 For each GFSA $\mathcal{A} = (M, \Sigma, T, s, F, \Delta)$ on a finite alphabet Σ, there is a standard FSA on alphabet Σ such that $\mathcal{A} \simeq \mathcal{A}'$.

Theorem 4.5 If $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{A}'$ is autonomous, then $\mathcal{A}$ is autonomous.

Corollary 4.6 A GFSA is autonomous if its skeleton is autonomous.

Based on Corollary 4.6 we have an error-trace guided symbolic analysis algorithm (procedure AnalyzeAutonomy in Figure 6). If the input GFSA is autonomous, AnalyzeAutonomy returns `null`; otherwise it returns the error trace which is a list of configurations. AnalyzeAutonomy starts from the input GFSA, and refines incrementally. During each cycle, AnalyzeAutonomy analyzes the skeleton of the current GFSA $\mathcal{A}'$. If the skeleton is autonomous, by Corollary 4.6, AnalyzeAutonomy simply returns and reports that the input GFSA is autonomous; otherwise, AnalyzeAutonomy identifies a pair of input/output transitions which start from the same state and lead to the violation the autonomy. For example, when analysis is applied to the skeleton of Figure 5(a), the two transitions starting at state 3 will be identified. Then procedure Refine is invoked to refine the current GFSA.

```
Procedure AnalyzeAutonomy($\mathcal{A}$): List
Begin
    $\mathcal{A}'$ = a copy of $\mathcal{A}$;
    While true do
        If skeleton of $\mathcal{A}'$ is autonomous Then return null;
        Find a pair $(s, (m_1, g_1), t_1), (s, (m_2, g_2), t_2)$ violating the autonomy
        ($\mathcal{A}'$,trace) = Refine($\mathcal{A}'$, $(s, (m_1, g_1), t_1), (s, (m_2, g_2), t_2)$);
        If trace $\neq$ null Then return trace;
    End While
End

Procedure Refine($\mathcal{A}$, $(s, (m_1, g_1), t_1), (s, (m_2, g_2), t_2)$) : (GFSA, List)
Begin
    If $(\text{Pre}(g_1) \wedge \text{Pre}(g_2))$ is satisfiable then
        Path = FindPath($\mathcal{A}$, $s$, $\text{Pre}(g_1) \wedge \text{Pre}(g_2)$)
        If Path $\neq$ null then return (null,Path);
    End If
    Let $\mathcal{A}' = (M', T', s'_0, F', \Delta')$ be a copy of $\mathcal{A}$;
    $T' = T' - \{s\} + \{s_1, s_2\}, F' = F' - \{s\} + \{s_1, s_2\}$;
    Substitute each $(t, (m_j, g_j), s)$ in $\Delta'$ with
        $(t, (m_j, g_j), s_1)$ and $(t, (m_j, g_j), s_2)$
    Substitute each $(s, (m_j, g_j), t)$ in $\Delta'$ s.t. $m_j \neq m_1$ and $m_j \neq m_2$ with
        $(s_1, (m_j, g_j), t)$ and $(s_2, (m_j, g_j), t)$
    Substitute $(s, (m_1, g_1), t_1)$ in $\Delta'$ with $(s_1, (m_1, g_1), t_1)$;
    Substitute $(s, (m_2, g_2), t_2)$ in $\Delta'$ with $(s_2, (m_1, g_1), t_2)$;
    Remove all unreachable transitions;
    return ($\mathcal{A}'$,null);
End
```

Figure 6. Refined Analysis of Autonomy

The input of Refine are two transitions (with guards g_1 and g_2 respectively) which leads to the violation of autonomy on the skeleton. Refine will try to refine the current GFSA by splitting the source state of these two transitions. If refinement succeeds, the refined GFSA is returned; otherwise, a concrete error trace is returned to show that the input GFSA is not autonomous. The first step of Refine is to compute the conjunction of the precondition of the two guards, i.e., $\text{Pre}(g_1) \wedge \text{Pre}(g_2)$. If the conjunction is satisfiable, it means that there is a possibility that at some configuration both transitions are enabled. Then we call procedure FindPath to find a concrete error trace, which will be explained later. In the case where the conjunction is not satisfiable, we can proceed to the refinement task. We split the source state of the two transitions into two states, and modify the transitions accordingly. Finally we eliminate transitions that cannot be reached during any execution of the GFSA.

Example 4.7 When procedure Refine is applied to Figure 5(a), and the two transitions starting at state 3, it first compute the conjunction of two preconditions: $\alpha.id \neq \beta.id \wedge \alpha.id = \beta.id$. Obviously the conjunction is not satisfiable. Then state 3 is split into two states, (state 3 and 4 in Figure 5(c)), and transitions are modified accordingly. Finally, unreachable transitions (dotted arrows in Figure 5(c)) are removed, and we get the GFSA in Figure 5(c). ▮

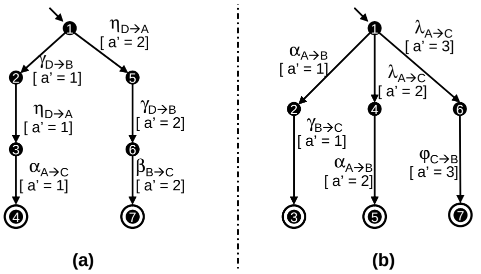


Figure 7. Counter Examples

The precondition operator Pre is standard operator in symbolic model checking, in which, first, all primed variables in the formula g are existentially quantified, and then they are eliminated using existential quantifier elimination. For example given a constraint g as "$a = 1 \wedge b' = 1$", its precondition is $\mathrm{Pre}(g) = \exists_{a'}\exists_{b'}(a = 1 \wedge b' = 1)$, which is equivalent to "$a = 1$".

Procedure $\mathrm{FindPath}$ has three inputs: a GFSA $\mathcal{A}$, a state s in $\mathcal{A}$, and a symbolic constraint g. $\mathrm{FindPath}$ will locate an error trace (a list of configurations) which starts from the initial state of $\mathcal{A}$, and finally reaches s in a configuration satisfying constraint g. The algorithm of $\mathrm{FindPath}$ is a variation of the standard symbolic model checking algorithm to explore reachable states. Details of $\mathrm{FindPath}$ can be found in [7]. Finally to reason about the correctness of the algorithm in Figure 6, the following Lemma is crucial.

Lemma 4.8 After procedure Refine successfully refines a GFSA $\mathcal{A}$, and let the result be $\mathcal{A}'$. The following is true: $L(\mathcal{A}') = L(\mathcal{A})$ and $\mathcal{A}' \simeq \mathcal{A}$.

Complexity of the algorithms in Figure 6 depends on the data domains associated with the input GFSA. When the message alphabet of a guarded conversation protocol is finite, algorithms in Figure 6 are guaranteed to terminate. For infinite domains, a constant loop limit can be used to terminate Procedure $\mathrm{FindPath}$ by force, which will result in a conservative analysis algorithm.

5. Symbolic Analysis of Lossless Join and Synchronous Compatibility

Now one natural question is: Can we apply a similar analysis algorithm to lossless join and synchronous compatibility conditions? The answer is negative, and the reason is explained below.

Example 5.1 Figure 4 is an example where the skeleton is lossless join, however the guarded conversation protocol is not. Figure 7(a) is an example where the protocol is lossless join, while its skeleton is not. There are four peers A, B, C, D in Figure 7(a), and all message classes consist of a single attribute a. In the beginning, peer D informs peer A and B about which path to take by the value of attribute a (1 for left branch or 2 for right branch). Then A and B knows who is going to send the last message (α or β), so there is no ambiguity. It can be verified that the protocol is lossless join. However the skeleton of Figure 7(a) is obviously not lossless join, because $\eta\gamma\alpha$ is included in its join closure. ▮

Example 5.2 If we make the message β in Figure 4 from peer C to A, the modified Figure 4 is an example which is not synchronous compatible, yet its skeleton is synchronous compatible. Figure 7(b) is another example, where the guarded conversation protocol is actually synchronous compatible however its skeleton is not, because after the partial conversation $\lambda\alpha$, peer B is ready to send γ however peer C is not receptive to it. ▮

The two examples above imply that we cannot conclude a guarded conversation protocol is lossless join or not based on the lossless join check on its skeleton. Same observation applies to synchronous compatibility and realizability as well. Due to the lack of results like Corollary 4.6, we do not have refined analysis for lossless join and synchronous compatibility. However, we do have a symbolic analysis algorithm for these two conditions, though the cost is greater. The analysis for lossless join is based on the following theoretical observation.

Recall that each GFSA $\mathcal{A}$ can be regarded as a transition system, and can be represented symbolically (details are given in [7]). Let $\mathcal{T}(\mathcal{A})$ denote the symbolic transition system derived from $\mathcal{A}$. From the initial configuration, we can compute all the reachable configurations of $\mathcal{T}(\mathcal{A})$, and let set of reachable configurations be $S^{\mathcal{A}}$. We have the following result.

Lemma 5.3 Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two guarded conversation protocols. The following statement is true:

$$(L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)) \Leftrightarrow (S^{\mathcal{A}_1} \wedge \mathcal{T}(\mathcal{A}_1) \Rightarrow S^{\mathcal{A}_2} \wedge \mathcal{T}(\mathcal{A}_2))$$

Lemma 5.3 naturally implies the symbolic analysis algorithm in Figure 8.

```
Procedure AnalyzeJoin($\mathcal{A}$): boolean
Begin
  Construct Cartesian product of projections of $\mathcal{A}$, let it be $\mathcal{A}'$.
  Compute $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$.
  return $S^{\mathcal{A}} \wedge \mathcal{T}(\mathcal{A}) \Rightarrow S^{\mathcal{A}'} \wedge \mathcal{T}(\mathcal{A}')$
End
```

Figure 8. Symbolic Analysis of Lossless Join

It is not hard to see that given a GFSA $\mathcal{A}$, when its alphabet is finite, algorithm shown in Figure 8 is decidable. When infinite domains are used, we can simply compute the approximate closure of $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$, and let them be S_1 and S_2 respectively. In Figure 8 we can replace $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$ with S_1 and S_2 respectively, which will give us a conservative algorithm.

Now we briefly discuss how to symbolically examine the synchronous compatibility. Recall the algorithm to check synchronous compatibility for a guardless conversation protocol. We project the protocol to each peer and determinize them, and then construct the Cartesian product from these deterministic projections. Then we check whether each state in the product is an illegal state (where some peer is not receptive to a message that another peer is ready to send). Note that determinization is a necessary step, otherwise the algorithm will not work. The symbolic analysis of synchronous compatibility for a guarded conversation protocol follows exactly the same procedure. However the key point here is how to "determinize" a GFSA. We present a symbolic "determinization" algorithm for GFSAs in [7].

6. Conclusions

This paper is an extension of the realizability analysis in [6] to guarded conversation protocols. We introduced a range of techniques to analyze each of the realizability conditions. There are several strategies of combining the use of them.

The most light-weight analysis is the skeleton analysis. However, this approach may not be very precise, hence, it may not recognize some realizable protocols. The most costly method is to combine the refined symbolic check of autonomy, the symbolic check of lossless join and synchronous compatibility. Note that the analysis procedure may be undecidable for infinite domains, however finite domains guarantee termination. Another combination is the use of skeleton analysis combined with refined analysis of autonomy. We can first check the autonomy of the skeleton. If the condition is not satisfied, we can apply the analysis until a concrete error trace is identified, or the autonomy is assured on a refined protocol. Then we can apply skeleton analysis on that refined protocol.

In the future, we plan to implement the algorithms described in this paper in our tool WSAT [14]. Symbolic model checking tool such as Composite Symbolic Library [17] can be used in implementing the presented symbolic analysis algorithms.

Acknowledgments

Bultan was supported in part by NSF Career award CCR-9984822 and NSF grant CCR-0341365; Fu was partially supported by NSF Career award CCR-9984822, NSF grants IIS-0101134 and CCR-0341365; Su was supported in part by NSF grants IIS-0101134 and IIS-9817432.

References

[1] Business Process Execution Language for Web Services (BPEL4WS), version 1.1. *available at* `http://www.ibm.com/developerworks/library/ws-bpel`.

[2] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. of the Twelfth Int. World Wide Web Conference (WWW)*, pages 403–410, May 2003.

[3] DAML Services Coalition. DAML-S: Semantic markup for web services. In *Proceedings of the Intl. Semantic Web Working Symposium (SWWS)*, July 30–August 1 2001.

[4] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. 18th IEEE Int. Conf. on Automated Software Engineering (ASE)*, 2003.

[5] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL Web Services. *To appear* in *Proceedings of 13th International Conference on World Wide Web (WWW)*.

[6] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. 8th Int. Conf. on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 188–200, 2003.

[7] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *UCSB Computer Science Dept. Technical Report (2004-10)*, 2004.

[8] J. E. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Proc. of 6th IEEE Int. Enterprise Distributed Object Computing Conference*, 2002.

[9] Java Message Service. http://java.sun.com/products/jms/.

[10] ITU-T recommendation Z.120, Message Sequence Charts (MSC'96), 1996.

[11] Microsoft Message Queuing Service. http://www.microsoft.com/msmq/.

[12] S. Narayanan and S. A. Mcllraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[13] W3C. Web Services Description Language (WSDL) version 1.1. *available at* `http://www.w3.org/TR/wsdl`.

[14] Web Service Analysis Tool (WSAT). `http://www.cs.ucsb.edu/~su/WSAT`.

[15] Web Service Choreography Interface (WSCI). `http://www.w3.org/TR/wsci/`.

[16] XML Schema. *available at* `http://www.w3c.org/XML/Schema`.

[17] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proceedings of the 7th International conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 52–66. Springer, April 2001.