# Realizability of Conversation Protocols with Message Contents

Xiang Fu, Tevfik Bultan, and Jianwen Su

A preliminary version of this paper was published as [11].

T. Bultan, and J. Su are with the Department of Computer Science, University of California at Santa Barbara, CA 93106, USA; X. Fu is with the School of Computer and Information Sciences, Geogia Southwestern State University, GA 31709, USA; xfu@canes.gsw.edu, {bultan, su}@cs.ucsb.edu

**Abstract**

A promising way to model the global behavior of a web service composition is to characterize the set of *conversations* among the participating web services. A conversation protocol specifies the desired global behaviors of a web service composition. The *realizability* problem is to decide whether, given a conversation protocol, a web service composition can be synthesized which generates exactly the same set of conversations specified by the protocol. This is a key problem in the top-down specification of the web service compositions. In our earlier work, we developed sufficient conditions for realizability of conversation protocols based on a model which abstracts the contents of the messages. The present paper extends our earlier work by allowing message contents to be used in the realizability analysis. We show that taking the message contents into account yields more accurate analysis. To overcome the state-space explosion caused by the message contents, we propose symbolic analysis techniques for the realizability conditions. In addition, we show that the analysis of one of the realizability conditions—the autonomy condition—can be done using an iterative refinement approach.

**Index Terms**

Web service composition, verification, conversation protocol, messages, realizability, asynchronous communication

## I. Introduction

Constructing highly dependable web service compositions is a challenging task because of their distributed nature. At the same time, without ensuring the dependability of the web service compositions it would not be advisable to use the web services framework in critical applications. Recently, automated verification of web service compositions has attracted significant attention in order to address this problem (e.g., [6], [8], and [20]). However, application of the automated verification techniques to web services is not trivial. One challenge is the establishment of a formal modeling and specification framework for web service compositions. There are numerous competing standards for composition of web services (e.g. BPEL4WS [2], WSCI [21], OWL-S [5]), which complicates this task. Additionally, certain characteristics of web services cause important problems. For example, asynchronous communication (supported by messaging platforms such as Java Message service (JMS) [17] and Microsoft Message Queuing service (MSMQ) [19]) may significantly increase the complexity of many verification problems, and in some cases make the problems undecidable [7].

An emerging paradigm of web service composition is to use "conversations" to describe interactions among participating web services ([13], [14], [1], [4], and [7]). A core idea common in these models is the use of finite state machines to represent some aspects of the global composition process. The state machines can involve two parties ([13] and [14]) or multi-parties ([4] and [7]), and may describe the global composition process directly ([13], [14], [4], and [7]) or may specify its local views ([1], [4], and [7]).

In our previous work reported in [4] and [7], we established a conversation oriented framework to specify web service compositions and reason about their global behaviors. Each participant (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 capture 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. Linear Temporal Logic (LTL) is naturally extended to this conversation based framework [7] and can be used to specify desired system goals such as "when a `cancel` request arrives, eventually the server should respond with a `cancel confirmation` message".

In general, there are two different ways to specify a web service composition:

1) The *bottom-up* approach (favored by many industry standards including WSDL [22] and BPEL4WS [2]) in which 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 [13] and Message Sequence Charts [18], [6]) in which the set of desired message events is specified and detailed peer implementations are left blank.

In [4] we generalized the notion of a conversation policy from two peers to arbitrary number of peers, and proposed the notion of a *conversation protocol*. Although the expressive power of a conversation protocol is weaker than that of the bottom-up approach, the top-down approach provides an advantage in that verification of composition properties can better utilize existing algorithms and tools developed in the verification community [4].

A conversation protocol is not always *realizable*, in the sense that there may not exist any peer implementations whose composition generates exactly the same set of conversations as specified by the protocol. In [7] we proposed sufficient conditions for realizability. The three *realizability conditions* proposed in [7] restrict control flows of a conversation protocol, and when these three conditions are satisfied, the projections of the protocol to each peer are guaranteed to be a realization of the protocol. One advantage of the realizability analysis is that it avoids the undecidability problem of verification that is inherent in the asynchronous communication (with queues) model for finite state machines [3].

Our realizability analysis results lead 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.

While the framework presented in [4] and [7] introduces the concepts of conversation protocols and realizability conditions, that model does not capture contents of messages (and thus the data semantics). As a result, the framework is too weak to be effective for analyzing and verification of practical web service applications. Naturally, it is interesting to raise the following question:

> *Can the realizability analysis in [7] be extended to a model that captures data semantics in conversation protocols?*

The main technical contributions of the present paper is to give a positive answer to the above question. We first extend the finite state (Mealy) model of web services to "guarded automata" over messages with contents. A message class in this extended model can have a finite set of attributes with static data types (integer, Boolean,

enumerated, or character). A guarded automaton extends a finite state machine with "guards" on the transitions; a guard is an expression that can allow/disallow a transition, and manipulate message contents (i.e., values of the message attributes). Similar to the approaches in [4] and [7], guarded automata can be used to model both conversation protocols and individual web services.

Given a conversation protocol represented by a guarded automaton, we define its "skeleton" to be the standard (i.e., guardless) version generated from the guarded protocol by removing all message contents and guards. One interesting question is whether the realizability of the skeleton implies the realizability of the original conversation protocol (with guards). This paper answers the above questions, in particular, we show that the realizability of the conversation protocols and their skeletons do not imply each other. However, we develop an additional realizability condition such that if a conversation protocol satisfies this extra condition and its skeleton is realizable, then the conversation protocol itself is also realizable. We also provide symbolic analysis techniques to relax the realizability conditions.

The remainder of the paper is organized as follows. Section 2 provides a brief review of the realizability analysis presented in [7] for content-less messages. Section 3 defines the notion of a guarded automaton and extends it to both conversation protocols and web services compositions. Section 4 discusses the conversion between the top-down (conversation protocols) and bottom-up (web service composition) specification approaches. Some of the procedures developed in Section 4 forms the basis for the symbolic analysis techniques described later. Section 5 presents some initial results for the realizability analysis of conversation protocols with finite domains. Section 6 introduces a light-weight skeleton analysis for conversation protocols. Section 7 includes the error-trace guided analysis for the autonomy condition which uses an iterative refinement strategy. Section 8 concludes the paper.

## II. Realizability Analysis for Conversation Protocols without Message Contents

In [7] we presented realizability analysis results for conversation protocols without message contents, i.e., the data semantics are abstracted away. In this section, we give a brief (and informal) review of the model and the realizability conditions in [7].

An important aspect of our model for web service composition is the asynchronous communication semantics. We assume that each peer has a FIFO message queue to store incoming messages. Each message queue is unbounded and reliable. When a message is sent it is inserted to the message queue of the receiver. The receiver can receive the message only when the message reaches the front end of the message queue. Intuitively, a "conversation" is the sequence of messages in the order they are sent. For example, assume that a service $A$ sends a message $\alpha$ to another service $B$, shortly after a third service $C$ sends a message $\beta$ to a fourth service $D$, and then all services terminate successfully. The conversation generated by this composition execution is $\alpha\beta$. Note that, the orderings of receive actions are not specified in a conversation. In other words, it cannot be inferred from the conversation $\alpha\beta$ that $B$ reads the message $\alpha$ before $D$ reads $\beta$. The receiving actions are completely determined by the individual services.

We start with the following three examples to motivate the discussion of realizability. Fig. 1 shows three

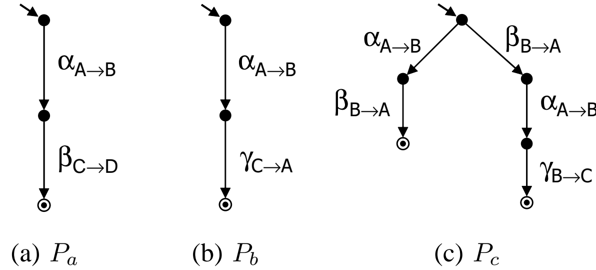


Fig. 1. Three Unrealizable Conversation Protocols

conversation protocols over four peers $A$, $B$, $C$, and $D$. The protocol $P_a$ involves all four peers, while the protocols $P_b$ and $P_c$ only involve $A$, $B$, and $C$. Each conversation protocol is shown as a finite state automaton (FSA) whose initial (denoted with an incoming edge with no source) and final (denoted with a double circle) states are explicitly identified. The language accepted by each FSA corresponds to the desired set of conversations specified by that conversation protocol. The subscript "$X \to Y$" for a message class denotes that the messages in that class are sent by the peer $X$ and received by the peer $Y$. Clearly, the protocols $P_a$ and $P_b$ allow only single conversations $\alpha\beta$, and respectively $\alpha\gamma$, while the protocol $P_c$ defines the following conversation set: $\{\alpha\beta, \beta\alpha\gamma\}$.

The three conversation protocols in Fig. 1 are not realizable. The conversation protocol $P_a$ is not realizable because any implementation of peers $A, B, C, D$ which generates the conversation $\alpha\beta$ will also generate the conversation $\beta\alpha$ since peer $C$ cannot know if $\alpha$ is sent or not before it sends $\beta$. Similarly, $P_b$ is not realizable since it does not allow the conversation $\gamma\alpha$ that can be generated by any implementation of the protocol. The protocol $P_c$ in Fig. 1(c) is more interesting. In an execution of this protocol, peers $A$ and $B$ can take the left and the right branches of the protocol (respectively), unaware that the other peer is taking a different branch. The following execution demonstrates this behavior: $A$ first sends the message $\alpha$, and $\alpha$ is stored in the queue of $B$; then $B$ sends the message $\beta$, and $\beta$ is stored in the queue of $A$. Peers $A$ and $B$ consume (i.e., receive) the messages in their respective queues, and finally $B$ sends the message $\gamma$. Hence the conversation $\alpha\beta\gamma$ is produced by the peer implementations. However, this conversation is not specified by the conversation protocol $P_c$ in Fig. 1(c).

In [7], we proposed three *realizability conditions* for conversation protocols (without message contents) which ensure realizability:

**1) Lossless join:** A conversation protocol satisfies the *lossless join* condition if its conversation set is equal to the *join* of its *projections* to each peer. Here the terms *join* and *projection* are borrowed from relational database theory. The *projection* of a conversation to a peer is the message sequence generated from the conversation by removing all unrelated messages to that peer. For example, consider the conversation $\alpha\beta$ defined by the protocol $P_a$ in Fig. 1(a). Its projection to peer $A$ is $\alpha$ (where message $\beta$ is removed from the conversation because $A$ is neither its sender nor its receiver), similarly, the projection of $\alpha\beta$ to $C$ is $\beta$. Given a set of languages (one language for each peer), the *join* of them includes all conversations whose projection to each peer is included in the language for that peer. For example, the join of the four languages $\{\alpha\}$, $\{\alpha\}$, $\{\beta\}$, $\{\beta\}$ (for $A$, $B$, $C$, and $D$ respectively) results in the

set $\{\alpha\beta, \beta\alpha\}$. This set contains all the conversations which when projected to $A$ and $B$ are included in the set $\{\alpha\}$, and when projected to $C$ and $D$ are included in the set $\{\beta\}$. A conversation protocol is said to to be a *lossless join* if its conversation set is the join of its projections to all peers. For example, if we project the conversation set $\{\alpha\beta\}$ of $P_a$ to all peers, we get the four languages $\{\alpha\}$, $\{\alpha\}$, $\{\beta\}$, $\{\beta\}$. When we construct the join of these languages, the resulting set is $\{\alpha\beta, \beta\alpha\}$. Hence, the protocol $P_a$ is not a lossless join, i.e., it does not satisfy the lossless join condition. However, it can be verified that the conversation protocols $P_b$ and $P_c$ in Fig. 1(b) and (c) satisfy the lossless join condition. It is shown in [9] that lossless join condition is a necessary condition for realizability.

**2) Synchronous compatibility:** A conversation protocol satisfies the *synchronous compatibility* condition if it can be implemented using synchronous communication semantics without generating a state in which a peer is ready to send a message while the corresponding receiver is not ready to receive that message. This condition can be checked as follows: i) Project the FSA describing the protocol to each peer by replacing all the transitions labeled with messages for which that peer is not the sender or the receiver with $\epsilon$-transitions (empty moves), ii) determinize the resulting automaton, iii) generate a product automaton by combining the determinized projections based on the synchronous communication semantics (i.e., corresponding send and receive transitions are taken simultaneously), and finally iv) check if there is a state in the product automaton where a peer is ready to send a message while the corresponding receiver is not ready to receive that message.

Consider the conversation protocol $P_b$ in Fig. 1(b). The projection of this protocol to peer $A$ does not change any of the transitions, whereas the projection to $B$ replaces the transition labeled with $\gamma$ by an $\epsilon$-transition, and the projection to $C$ replaces the transition labeled with $\alpha$ by an $\epsilon$-transition. After the determinization, the automaton for peer $C$ consists of one initial and one final state, and a single transition labeled $\gamma$ from the initial state to the final state. When we generate the automaton which is the product of the projections, in the initial state of the product automaton, peer $C$ is ready to send the message $\gamma$ but the receiver, i.e., $A$, is not yet ready to receive the message. Hence, the conversation protocol $P_b$ does not satisfy the synchronous compatibility condition. On the other hand, the conversation protocols $P_a$ and $P_c$ in Fig. 1(a) and (c) satisfy the synchronous compatibility condition.

**3) Autonomy:** A conversation protocol satisfies the *autonomy condition* if at every state, each peer is able to do exactly one of the following: send a message, receive a message, or terminate. Note that a state may have multiple transitions corresponding to different send operations for a peer, and this does not violate the autonomy condition. Similarly, a peer can have multiple receive transitions from the same state. However, the autonomy condition is violated if there are two (or more) transitions originating from a state such that one of them is a send operation and another one is a receive operation for the same peer. For example, consider the conversation protocol $P_c$ in Fig. 1(c). The two transitions labeled $\alpha$ and $\beta$ originating from the initial state violate the autonomy condition. The transition labeled $\alpha$ corresponds to a send operation for $A$, whereas the transition labeled $\beta$ corresponds to a receive operation for $A$. The conversation protocols $P_a$ and $P_b$ in Fig. 1(a) and (b) satisfy the autonomy condition.

In [7] we showed that a conversation protocol (without message content) is realizable if it satisfies all three realizability conditions listed above. We also showed that if a conversation protocol satisfies these three realizability

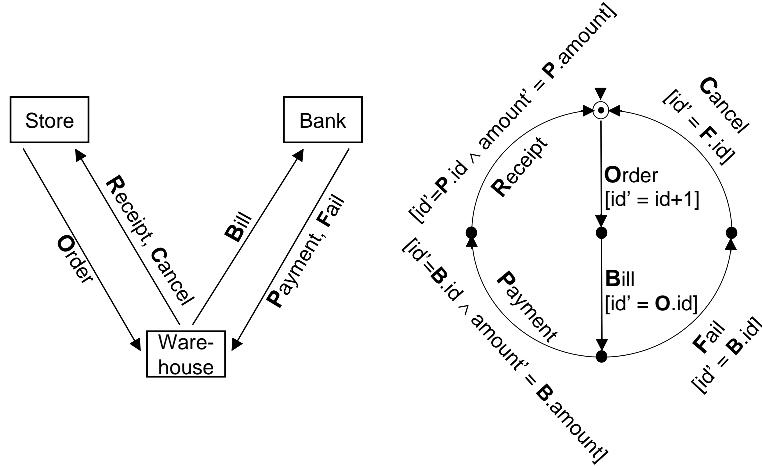


Fig. 2. The Composition Schema and the Conversation Protocol for a Simplified Warehouse Example

conditions, its projections to the peers will generate the same conversation set specified by the protocol.

## III. The Guarded Automata Model

In this section we present a formal model, called the "guarded automata" model, for conversations with message contents. As we mentioned earlier, composite web services can be specified using either the top-down or the bottom-up approach. In this section, both specification approaches are based on the guarded automata, which allow message classes to have contents and use guards to manipulate data semantics. The guarded automata model subsumes the finite state automata or Mealy model (which hides message contents) used in [4], [7]. In fact, if the messages have no contents and all the guards in a guarded automaton are set to true then a guarded automaton is actually a finite state (Mealy) machine. We begin this section by defining the composition schema, then we present the formal models for the top-down and bottom-up specification approaches, namely, conversation protocols and web service compositions.

### A. Composition Schema

In this paper, we assume that once the composition of a set of web services is determined, the structure of composition is fixed from then on. We use the notion "composition schema" to define the structure of a composition. A composition schema includes the information about peers, message classes, and the message domains for each message class.

*Definition 3.1:* A *composition schema* is a tuple $(P, M, \Sigma)$ where $P$ is a finite set of peers, $M$ is a finite set of message classes, and $\Sigma$ is the set of messages. Each message class $c \in M$ has a finite set of attributes which has a static data type. A message $m \in \Sigma$ is an instance of message class. Thus the message alphabet $\Sigma$ can be defined as follows:

$$\Sigma = \bigcup_{c \in M} \{c\} \times \text{DOM}(c),$$

where $\text{DOM}(c)$ is the domain of the message class $c$. Notice that $\Sigma$ may be a finite or infinite set, which is determined by the domain of message classes.

*Example 3.1:* The diagram on the left side of Fig. 2 illustrates a composition schema which consists of three peers, Store, Bank, and Warehouse. Instances of message classes such as **O**rder, **B**ill, and **P**ayment, are transmitted among these three peers. In the remainder of this section, we assume that the message classes **B**ill, **P**ayment, and **R**eceipt have two integer attributes id and amount, and all other message classes in Fig. 2 have a single integer attribute id. A message, i.e., an instance of a message class, is written in the following form: "class(contents)". For example, $\mathbf{B}(100, 2000)$ stands for a **B**ill message whose id is 100 and amount is 2000. Here **B**ill is represented using its capitalized first letter **B**. Obviously for message class **R**eceipt, its domain is $N \times N$, where $N$ is the set of integers. ∎

### B. Conversation Protocol

Given a composition schema, the design of a web service composition can be specified using a conversation protocol, in a top-down fashion. A conversation protocol is a guarded automaton, which specifies the desired global behaviors of the web service composition. Its formal definition is as below.

*Definition 3.2:* A *conversation protocol* is a tuple $\langle (P, M, \Sigma), \mathcal{A} \rangle$, where $(P, M, \Sigma)$ is a composition schema, and $\mathcal{A}$ is a guarded automaton (GA) defined in the following. The guarded automaton $\mathcal{A}$ is a tuple $(M, \Sigma, T, s, F, \delta)$ where $M$ and $\Sigma$ 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 $\delta$ is the transition relation. Each transition $\tau \in \delta$ is in the form of $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of $\tau$, $c \in M$ is a message class and $g$ is the *guard* of the transition. During a run of a guarded automaton, a transition is taken only if the guard evaluates to true.

Formally a guard $g$ is a predicate of the following form:

$$g(\text{ATTR}(c'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$$

where $\text{ATTR}(c')$ are the attributes of the message that is being sent, and $\text{ATTR}(M_i^{in} \cup M_i^{out})$ are the attributes of the *latest* instances of the message classes that are received or sent by peer $p_i$, where $p_i$ is the sender of message class $c$ (if for a message class there is no instance received or sent yet, then attribute values for that message class are undefined). For example, in Fig. 2, the guard of the transition to send **O**rder is: **O**rder.id' $=$ **O**rder.id $+ 1$. The guard express the semantics to increment the value of the attribute id by $1$ whenever a new **O**rder message is sent. Notice that in a transition guard, the primed form of an attribute refers to the value of the attribute in the message that will be sent, and the non-primed form refers to the value of the attribute in the latest instance of the corresponding message. With the use of primed forms of variables, the "=" symbols in guards stand for the "equality" instead of "assignment". Thus declarative semantics (instead of procedural semantics) here allows convenient application of model checking tools.

*Example 3.2:* By studying the semantics of transition guards in Fig. 2, it is not hard to see that the conversation protocol in Fig. 2 describes the following desired message exchange sequence of the simplified warehouse example: an **O**rder is sent by Store to Warehouse, and then Warehouse sends a **B**ill to Bank. The Bank either responds with a **P**ayment or rejects with a **F**ail message. Finally Warehouse issues a **R**eceipt or a **C**ancel message. The guards determine the contents of the messages. For example, the id and amount of each **P**ayment must match those of the latest **B**ill message. ▮

The language accepted by a guarded automaton can be naturally defined by extending standard finite state automaton semantics. Given a GA $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$, a *run* of $\mathcal{A}$ is a path which traverses from the initial state $s$ to a final state in $F$. A message sequence $w \in \Sigma^*$ is accepted by $\mathcal{A}$ if there exists a corresponding run. For example, it is not hard to infer that the following is a word accepted by the GA in Fig. 2:

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

Given a conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, its language is defined as $L(\mathcal{P}) = L(\mathcal{A})$.

Notice that based on the definition of guards, a conversation protocol is only able to remember the attributes of the last sent message for each message class. We do not think this is an important restriction based on the web services we studied. More information about the sent and received messages can be stored in the states of the conversation protocol. Another approach would be to extend the guard definition so that the guards can refer to the last $\ell$ instances of each message class where $\ell$ is a fixed integer value. Such an extension would not effect the results we will discuss in the following sections.

### C. Web Service Composition

Bottom-up specified web service compositions is also built on the GA model, however, the GA used to describe a peer is slightly different than the one used to describe a conversation protocol. The formal definition is given below.

*Definition 3.3:* A *web service composition* is a tuple $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where $(P, M, \Sigma)$ is the composition schema, $n = |P|$, and for each $i \in [1..n]$, $\mathcal{A}_i$ is the peer implementation for $p_i \in P$. Each $\mathcal{A}_i$ is a tuple $(M_i, \Sigma_i, t_i, s_i, F_i, \delta_i)$ where $M_i$, $\Sigma_i$, $t_i$, $s_i$, $F_i$, and $\delta_i$ denote the set of message classes, set of messages, set of states, initial state, final states, and transition relation, respectively. A transition $\tau \in \delta_i$ can be one of the following three types: a send transition $(t, (!\alpha, g_1), t')$, a receive transition $(t, (?\beta, g_2), t')$, and an $\epsilon$-transition $(t, (\epsilon, g_3), t')$, where $t, t' \in T_i$, $\alpha \in M_i^{out}$, $\beta \in M_i^{in}$, and $g_1$, $g_2$, and $g_3$ are predicates in the forms $g(\text{ATTR}(\alpha'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$, $g(\text{ATTR}(\beta'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$, and $g(\text{ATTR}(M_i^{in} \cup M_i^{out}))$, respectively.

The send and receive transitions are denoted using symbols “!” and “?”, respectively. In an $\epsilon$-transition, the guard determines if the transition can take place based on the contents of the latest message for each message class related to that peer. For a receive transition $(t, (?\beta, g), t')$, its guard determines whether the transition can take place based on the contents of the latest messages (i.e., the $\text{ATTR}(M_i^{in} \cup M_i^{out})$ in the formula of $g$) as well as the message at the queue head (i.e., the $\text{ATTR}(\beta')$). Notice that, even if the message at the queue is of class $\beta$, it is still possible that

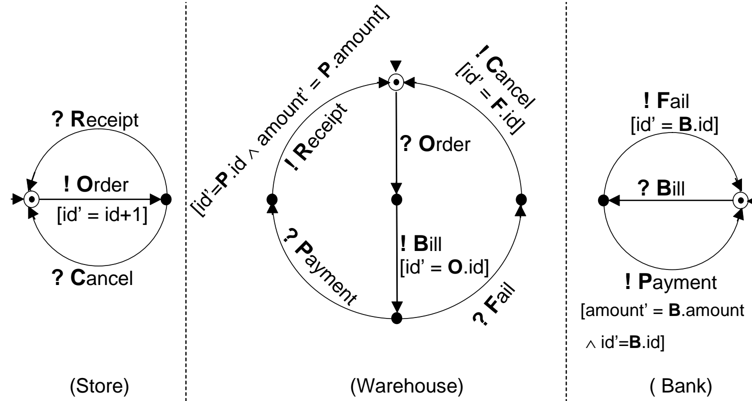


Fig. 3. A Realization of the Conversation Protocol in Fig. 2

the peer gets stuck because the contents of the queue head might not satisfy the predicate $g$. For a send transition $(t, (!\alpha, g), t')$, the guard $g$ determines not only the transition condition but also the contents of the message being sent (i.e., the $\text{ATTR}(\alpha')$). For example, Fig. 3 shows a web service composition which realizes the conversation protocol in Fig. 2. Note that, if the guard for a transition is not shown, then it means that the guard is "`true`".

The conversations produced by a web service composition can be defined similarly as the language accepted by a guarded automaton. However, one important difference is that the effect of queues has to be taken into account.

To characterize the formal semantics of a conversation, we need to define the notion of *global configuration* of a web service composition. A global configuration is used to capture a "snapshot" of the whole system. Generally, a global configuration contains the information about the local state, queue contents for each peer, as well as the latest sent and received copies for each message class (s.t. guards can be evaluated). The *initial configuration* of a web service composition is obviously the one where each peer is in its initial state, each FIFO queue is empty, and the latest copies for each message class are the constant "undefined value". Similarly, a *final global configuration* is a configuration where each peer stays in a final state, and each peer queue is empty. Then a *run* of a web service composition can be defined as a sequence of global configurations, which starts from the initial configuration and ends with a final configuration. Between each pair of neighboring configurations $c_i$ and $c_{i+1}$ in a complete run, $c_i$ evolves into $c_{i+1}$ by taking one action by a peer. This action can be a send, receive (from queue), or $\epsilon$ action which corresponds to a transition in that peer GA. Obviously, for this action to take place, the associated guard of that transition must be satisfied. A *conversation* is a message sequence, for which there is a corresponding run.

Now given both the definitions of conversation protocols and web service compositions, we can define the notion of *realizability* that relates them.

DEFINITION. Let $\mathcal{C}(\mathcal{S})$ denote the set of conversations of a web service composition $\mathcal{S}$. We say $\mathcal{S}$ *realizes* a conversation protocol $\mathcal{P}$ if $\mathcal{C}(\mathcal{S}) = L(\mathcal{P})$.

# IV. Conversion Between Top-down and Bottom-Up Specifications

Since a composite web service has two specification approaches: top-down specified conversation protocol, and bottom-up specified web service composition. One interesting question is: can we convert from one specification approach to the other? In this section we present two related operations, namely "*projection*" and "*Cartesian product*". The projection of a conversation protocol defines a web service composition, and the Cartesian product of a web service composition defines a conversation protocol. However, notice that the projection of a top-down conversation protocol is not guaranteed to generate exactly the same set of conversations as specified by the protocol. Similarly, the Cartesian product of a bottom-up web service composition is not guaranteed to specify all the possible conversations that can be generated by that web service composition. Only when additional sufficient conditions (presented in Section 4) are satisfied, can projection and Cartesian product be used to convert from one specification approach to the other, without loss of semantics. Another interesting observation is that projection in the GA model is more complex than that of the standard FSA model. This makes the symbolic realizability analysis much more challenging in the GA model.

## A. Cartesian Product

Let $\mathcal{S} = (\langle(P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$ be a web service composition, where for each $i \in [1..n]$, $\mathcal{A}_i$ is a tuple $(M_i, \Sigma_i, T_i, s_i, F_i, \delta_i)$. The *Cartesian product* of all peers in $\mathcal{S}$ is a GA $\mathcal{A}' = (M, \Sigma, T', s', F', \delta')$, where each state $t' \in T'$ is a tuple $(t_1, \ldots, t_n)$, such that for each $i \in [1..n]$, $t_i$ is a state of peer $\mathcal{A}_i$. The initial state $s'$ of $\mathcal{A}'$ corresponds to the tuple $(s_1, \ldots, s_n)$, and a final state in $F'$ corresponds to a tuple $(f_1, \ldots, f_n)$ where for each $i \in [1..n]$, $f_i$ is a final state of $\mathcal{A}_i$. Let $\rho$ map each state $t' \in T'$ to the corresponding tuple, and further let $\rho(t')[i]$ denote the $i$'th element of $\rho(t')$. For each pair of states $t$ and $t'$ in $T'$, a transition $(t, (m, g'), t')$ is included in $\delta'_i$ if there exists two transitions $(t_i, (!m, g_i), t'_i) \in \delta_i$ and $(t_j, (?m, g_j), t'_j) \in \delta_j$ such that

1) (sending and receiving peers take the send and receive transitions simultaneously) $\rho(t)[i] = t_i$, $\rho(t')[i] = t'_i$, $\rho(t)[j] = t_j$, $\rho(t')[j] = t'_j$, and for each $k \neq i \wedge k \neq j$, $\rho(t')[k] = \rho(t)[k]$, and
2) (both guards need to hold) $g' = g_i \wedge g_j$, and $g'$ is satisfiable.

Clearly, by the above definition, the construction of the Cartesian product can start from the initial state of the product (which corresponds to the tuple of initial states of all peers), then iteratively include new transitions and states. From a state in the product, a transition is added to point to another destination state, only if there is a pair of peers which execute the corresponding send and receive actions simultaneously, from the source state. Obviously, the construction can always terminate because the number of transitions and states of all peers is finite. However, different than the Cartesian product construction for the FSA model, the algorithm here requires the ability to decide the satisfiability of symbolic constraints. In addition, to apply Cartesian product, none of the peer implementation can have $\epsilon$ transitions. We will present the algorithm to remove $\epsilon$ transitions later in this section.

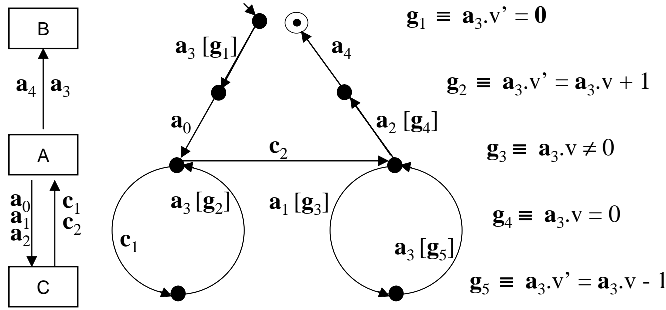


Fig. 4. The Infinite Content (IC) Conversation Protocol for Example 4.1

### B. Projection

Projection of a GA conversation protocol is complex and interesting. For example, a GA conversation protocol with an infinite message alphabet may not always have an "exact" projection. However, if its message alphabet is finite, a GA conversation protocol is always guaranteed an "exact" projection, although the cost may be high. We introduce the "exact" projection algorithm for GA conversation protocols with finite message alphabet. Several "coarse" algorithms are also presented, where accuracy is traded for efficiency. Notice that the coarse algorithms work for both infinite and finite message alphabet cases.

*Definition 4.1:* A conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ is called an *Infinite Content (IC)* conversation protocol if $\Sigma$ is an infinite set; otherwise $\mathcal{P}$ is an *Finite Content (FC)* conversation protocol. Similarly a guarded automata is either an IC-GA or FC-GA and a web service composition is either an IC composition or a FC composition.

In the following, we show that an IC conversation protocol may not have an exact projection, i.e., when its conversation set is projected to a peer, the projection may not be a regular language.

*Example 4.1:* Fig. 4 presents an IC conversation protocol (let it be $\mathcal{P}$) which involves three peers $A$, $B$, $C$. Message class $\mathbf{a}_3$ contains a single infinite-domain integer attribute v (used as an counter), and other message classes do not have message contents. The desired message exchange sequence, as described by $\mathcal{P}$, proceeds as follows. Initially peer $A$ sends an $\mathbf{a}_3$ message, and initializes its attribute v as $0$. Then $A$ sends $\mathbf{a}_0$ to request $C$ to start sending $\mathbf{c}_1$ messages. For each $\mathbf{c}_1$ from $C$, peer $A$ sends one $\mathbf{a}_3$ to B, and increments the counter of $\mathbf{a}_3$ by 1. Peer $C$ may nondeterministically decide when to end the sending of $\mathbf{c}_1$, and the message $\mathbf{c}_2$ is used to mark this end. After message $\mathbf{c}_2$ is received, peer $A$ starts to send a sequence of $\mathbf{a}_1$ messages to $C$. The number of $\mathbf{a}_1$ messages will be exactly the same as $\mathbf{c}_1$. This is implemented by decrementing the counter of $\mathbf{a}_3$ message until it reaches the value $0$. Hence, each conversation $w \in L(\mathcal{P})$ must be of the following form, where $n$ is the number of $\mathbf{c}_1$ and $\mathbf{a}_1$ being sent in the conversation:

$$\mathbf{a}_3\mathbf{a}_0(\mathbf{c}_1\mathbf{a}_3)^n\mathbf{c}_2(\mathbf{a}_1\mathbf{a}_3)^n\mathbf{a}_2\mathbf{a}_4.$$

It is not hard to see that the projection of $L(\mathcal{P})$ to peer $C$, i.e., $\pi_C(L(\mathcal{P}))$ is the set $\{\mathbf{a}_0{\mathbf{c}_1}^n\mathbf{c}_2{\mathbf{a}_1}^n\mathbf{a}_2 \ \mid n \geq 0\}$,

**Procedure** ProjectGA($\langle(P, M, \Sigma), \mathcal{A}\rangle$, $i$): GA
**Begin**
**Let** $\mathcal{A}' = (M, \Sigma, T, s_0, F, \delta)$ be a copy of $\mathcal{A}$.
Substitute each $(q_1, (a, g_1), q_2)$ where $a \notin M_i$ with $(q_1, (\epsilon, g_1'), q_2)$.
Substitute each $(q_1, (a, g_2), q_2)$ where $a \in M_i^{in}$ with $(q_1, (?a, g_2'), q_2)$.
Substitute each $(q_1, (a, g_3), q_2)$ where $a \in M_i^{out}$ with $(q_1, (!a, g_3'), q_2)$.
// The generation of $g_1'$, $g_2'$ and $g_3'$ uses either **Coarse Processing 1** or **Coarse Processing 2**.
// **Coarse Processing 1:**
// $g_1' = g_2' =$"true", and $g'_3 = g_3$.
// **Coarse Processing 2:**
// $g_1'$, $g_2'$ are the predicates generated from $g_1$, $g_2$ (resp.),
// by eliminating unrelated message attributes via existential quantification.
// $g_3' = g_3$.
**return** $\mathcal{A}'$.
**End**

Fig. 5. Coarse Projection of Guarded Automata

which is obviously a context free language. However, since none of the messages (sent or received) by peer $C$ have message contents, any GA with the alphabet of $C$ is essentially a standard FSA, and it cannot accept $\pi_C(L(\mathcal{P}))$, which is not regular. ∎

For FC conversation protocols, it is always possible to construct a corresponding projected composition $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$ where each peer implementation of $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$ is an "exact" projection of $\mathcal{P}$. Formally, the result is presented in the following theorem.

*Theorem 4.2:* For each FC conversation protocol $\mathcal{P}$, there exists an FC web service composition $\mathcal{S} = \langle(P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$ where for each $i \in [1..n]$: $L(\mathcal{A}_i) = \pi_i(L(\mathcal{P}))$.

*Proof:* Let $\mathcal{P} = \langle(P, M, \Sigma), \mathcal{A}\rangle$ where $n = |P|$. Since $\Sigma$ is finite, we can easily construct a standard FSA $\mathcal{A}'$ from $\mathcal{A}$ such that $L(\mathcal{A}') = L(\mathcal{A})$, where the alphabet of $\mathcal{A}'$ is the message alphabet $\Sigma$ of $\mathcal{A}$. The construction of $\mathcal{A}'$ is essentially an exploration of all reachable global configurations of $\mathcal{A}$. Again, notice that since $\Sigma$ is finite, the number of global configurations of $\mathcal{A}$ is finite, hence the size of $\mathcal{A}$ is finite. Now project $\mathcal{A}'$ to each peer, and let them be $\mathcal{A}_1', \ldots, \mathcal{A}_n'$, respectively. Obviously, each of the standard FSA $\mathcal{A}_1', \ldots, \mathcal{A}_n'$ can be converted into an equivalent GA by associating dummy guards with each transition. ■

Based on Theorem 4.2, we have the following definition for exact projection.

*Definition 4.2:* Given an FC conversation protocol $\mathcal{P}$, let $\mathcal{S} = \langle(P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$ be the corresponding web service composition given in the proof of Theorem 4.2. $\mathcal{S}$ is called the *(exact) projected composition* of $\mathcal{P}$, written as $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$.

The construction of $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$ (as shown in Theorem 4.2), however, can be very costly—it requires essentially a reachability analysis of the state space of the FC conversation protocol. In Fig. 5, we present a light-weight projection algorithm, which is not "exact", but works for both FC and IC conversation protocols.

Given the input of GA protocol and the peer to project to, the *coarse projection* algorithm in Fig. 5 simply replaces each unrelated transition with $\epsilon$-transitions, and adds "!" and "?" for send and receive transitions respectively. The algorithm provides two different levels of "coarse processing". In **Coarse Processing 1**, the guards of the $\epsilon$-transitions and receive transitions are essentially dropped (by setting them to "true"), and the guards of the send transitions remain the same. In **Coarse Processing 2**, existential quantification is used to eliminate the unrelated message attributes from guards. The following example illustrate the point of existential quantification.

*Example 4.3:* Given a GA conversation protocol on three peers $p_1$, $p_2$ and $p_3$. Let $\tau = (t, (m_1, g), t')$ be a transition in the protocol, where $m_1 \in M_3^{out} \cap M_1^{in}$, $m_2 \notin M_1$, and

$$g \equiv m_1.\mathsf{id} + m_2.\mathsf{id} < 3 \ \wedge \ m_2.\mathsf{id} > 0.$$

We assume that $m_1.\mathsf{id}$ and $m_2.\mathsf{id}$ are both of integer type. During the projection to peer $p_1$, if $\tau$ is being processed using **Coarse Processing 1**, the corresponding transition would be $(t, (?m_1, true), t')$; if **Coarse Processing 2** is used, the corresponding transition would be $(t, (?m_1, g'), t')$ where $g' \equiv \exists_{m_2.\mathsf{id}}\ g$, and after simplification, $g' \equiv m_1.\mathsf{id} < 2$. ∎

Given a GA conversation protocol $\mathcal{P}$, the web service composition generated using coarse-1 and coarse-2 processing algorithms are denoted as $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$, respectively. The following theorem reveals the relationship between $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$.

*Lemma 4.4:* Given a conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, and its projections $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$. For each $1 \leq i \leq |P|$, let $\mathcal{A}_i^{\mathbb{C}1}$ and $\mathcal{A}_i^{\mathbb{C}2}$ be the peer implementation of $p_i$ in $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$ (resp.). Then, the following holds: $\pi_i(L(\mathcal{P})) \subseteq L(\mathcal{A}_i^{\mathbb{C}2}) \subseteq L(\mathcal{A}_i^{\mathbb{C}1})$.

*Proof:* For each transition $(t, (m, g), t')$, let $g_1$ and $g_2$ be the corresponding guards generated by Coarse Processing 1 and Coarse Processing 2. Obviously, $g \Rightarrow g_2 \Rightarrow g_1$, where " $\Rightarrow$ " is the boolean implication operator. This fact immediately leads to the lemma. ■

*C. Determinization of Guarded Automata*

We now introduce a "determinization" algorithm for guarded automata, which is useful in the decision procedures for realizability conditions. The "determinization" process consists of two steps: (1) eliminate $\epsilon$-transitions from a guarded automaton, and (2) determinize the resulting GA from step (1). The $\epsilon$-transition elimination algorithm is presented in Fig. 6, which is an extension of the $\epsilon$-transition elimination for standard FSA.

One interesting part of the algorithm in Fig. 6 is the collection and handling of guards (lines 10 to 13). Note that the transition $(t, (m, g'), t'')$ in line 11 is a replacement for a set of paths, where each path is a concatenation of the transition $(t, (m, g), t')$ and an $\epsilon$-path from $t'$ to $t''$. It is not hard to see that the guard $g'$ should be the conjunction of $g$ and $g''$ where $g''$ is the disjunction of the conjunctions of guards along each $\epsilon$-path from $t'$ to $t''$. Note that for each $\epsilon$-transition, its guard is only a "transition condition" which does not affect the message instance vector in a GA configuration. Hence it suffices to consider those non-redundant paths, and obviously the number of non-redundant paths is finite. Thus the algorithm in Fig. 6 will always terminate.

1. **Procedure** ElimGA ($\mathcal{A}$): GA
2. **Begin**
3. **Let** $\mathcal{A}' = (M, \Sigma, T, s, F, \delta)$ be a copy of $\mathcal{A}$.
4. **For each** $t \in T$ **Do**
5. insert each $t'$ that is reachable from $t$ via $\epsilon$-paths in $\epsilon$-closure($t$).
6. **End For**
7. // let $\Upsilon(t, t')$ be the set of non-redundant $\epsilon$-paths from $t$ to $t'$.
8. // each transition in $\delta$ appears at most once in a non-redundant path.
9. // let $\mathrm{cond}(\ell)$ be the conjunction of all guards along a non-redundant path $\ell$.
10. **For each** transition $(t, (m, g), t') \in \delta$ **do**
11. insert a transition $(t, (m, g'), t'')$ in $\delta$ for each $t''$ in $\epsilon$-closure($t'$),
12. where $g' = g \wedge g''$ and $g'' = \bigvee_{\ell \in \Upsilon(t', t'')} \ \mathrm{cond}(\ell)$.
13. **End For**
14. eliminate all $\epsilon$-transitions from $\delta$.
15. **return** $\mathcal{A}'$.
16. **End**

Fig. 6. $\epsilon$-transitions Elimination for Guarded Automata

1. **Procedure** DeterminizeGA($\mathcal{A}$): GA
2. **Begin**
3. **Let** $\mathcal{A}' = (M, \Sigma, T, s, F, \delta)$ be a copy of ElimGA($\mathcal{A}$).
4. Mark all states in $T$ as "unprocessed".
5. **For each** "unprocessed" state $t \in T$ **Do**
6. **For each** message class $m \in M$ **Do**
7. Let $\{\tau_1, \ldots, \tau_k\}$ include each transition $\tau_i = (t, (m, g_i), t'_i)$ which starts from $t$ and sends $m$.
8. mark each $\tau_i$ as "toRemove"
9. **For each** $c = \ell_1 \wedge \cdots \wedge \ell_k$ where $\ell_i$ is $g_i$ or $\bar{g_i}$ **Do**
10. **If** $c$ is satisfiable **Then**
11. **Let** $s'$ be a new state name, include $s'$ in $T$.
12. include $(t, (m, c), s')$ in $\delta$.
13. **For each** $j \in [1..k]$ s.t. $g_j$ (instead of $\bar{g_j}$) appears in $c$ **Do**
14. **For each** transition $\tau' = (t'_i, (m', g'), t''_j)$ from $t'_i$ **do**
15. include $(s', (m', g'), t''_j)$ in $\delta$
16. mark $\tau'$ as "toRemove". mark $t'_i$ as "toRemove".
17. **End For**
18. **End For**
19. **End If**
20. **End For**
21. **End For**
22. remove all states and transitions that are marked as "toRemove".
23. **End For**
24. return $\mathcal{A}'$.
25. **End**

Fig. 7. Determinization of Guarded Automata

Fig. 7 presents the determinization algorithm for a Guarded Automaton. Note that the idea of the algorithm is rather different than the determinization algorithm for a standard FSA. The key idea of the algorithm is the part

shown in lines 9 to 20, where for each state and each message class, we collect all transitions for that message class, enumerate every combination of guards, and generate a new transition for that combination. For example, suppose at some state $t$, two transitions are collected for message class $m$, let their guards be $g_1$ and $g_2$ respectively. Four new transitions will be generated for (resp.), and the two original transitions are removed from the transition relation. It is not hard to see that for each word $w \in L(\mathcal{A}')$, where $\mathcal{A}'$ is the resulting GA, there exists one and only one run for $w$, due to the enumeration of the combinations of guards. Since the procedure of enumerating guards and reassembling states does not deviate from the semantics of the original guards(e.g., $g_1 \wedge g_2 \vee g_1 \wedge \bar{g_2} \vee \bar{g_1} \wedge g_2 \vee \bar{g_1} \wedge \bar{g_2} = g_1 \vee v_2$), each $\mathcal{A}$ is equivalent to its determinization $\mathcal{A}'$ (after applying DeterminizeGA), i.e., $L(\mathcal{A}) = L(\mathcal{A}')$.

## V. Realizability Analysis of Finite Content Conversation Protocols

We now concentrate on the realizability analysis for the simplest case, the Finite Content (FC) conversation protocols. The results directly follow the standard FSA case [7], [9]. We will use three sufficient conditions to restrict the control flow of an FC conversation protocol. To introduce the first realizability condition, we need to define the notion of operation *join* formally.

Given $n$ peers and $n$ 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:

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

where $\pi_i(w)$ denotes the projection of the word $w \in \Sigma^*$ to $\Sigma_i$ as defined earlier. For a language $L \subseteq \Sigma^*$, its *join closure* is defined as: $\text{JOINC}(L) = \bowtie (\pi_1(L), ..., \pi_n(L))$.

*Definition 5.1:* A conversation protocol $\mathcal{P}$ satisfies the *lossless join* condition if $L(\mathcal{P}) = \text{JOINC}(L(\mathcal{P}))$.

Generally the lossless join condition requires a conversation protocol to be complete, in the sense that, when we construct the join of its projections to all peers, the join should be exactly the same as the protocol. The second condition, i.e., the synchronous compatibility condition, is defined as follows.

*Definition 5.2:* Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ be a conversation protocol, and $n = |P|$. $\mathcal{P}$ is said to be *synchronous compatible* if for each word $w \in \Sigma^*$ and each message $\alpha \in \Sigma_a^{\text{out}} \cap \Sigma_b^{\text{in}}$ for $a, b \in [1..n]$, the following holds:

$$(\forall i \in [1..n],\ \pi_i(w) \in \pi_i(L_{\preceq}^*(\mathcal{A})))\ \wedge\ \pi_a(w\alpha) \in \pi_a(L_{\preceq}^*(\mathcal{A}))$$
$$\Rightarrow \quad \pi_b(w\alpha) \in \pi_b(L_{\preceq}^*(\mathcal{A})).$$

Here $L_{\preceq}^*(\mathcal{A})$ is the prefix language of $\mathcal{A}$, i.e., $L_{\preceq}^*(\mathcal{A}) = \{w \mid w \text{ is a prefix of } w', \text{ and } w' \in L(\mathcal{A})\}$. Intuitively, the condition requires that, if we use the determinized projections of the protocol as peer implementations, when these projections are composed synchronously, at any time (where $w$ is the sequence of message exchanged), if a peer $a$ wants to send a message $\alpha$, then peer $b$ is in the state ready to accept $\alpha$. Otherwise, the property is violated. The third condition, i.e., autonomy condition, is defined as follows.

*Definition 5.3:* A conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ is *autonomous* if for each peer $p_i \in P$ and for each finite prefix $w \in L_{\preceq}^*(\mathcal{A})$, at most one of the following three conditions hold: a) the next transitions of $p_i$ (including

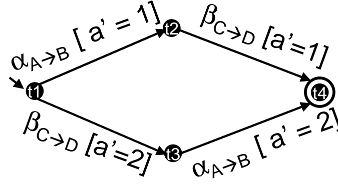


Fig. 8. The Conversation Protocol for Example 6.1

transitions that are reachable through $\epsilon$-transitions) are all send operations, b) the next transitions of $p_i$ (including transitions that are reachable through $\epsilon$-transitions) are all receive operations, or c) $p_i$ is in a final state.

*Theorem 5.1:* An FC conversation protocol $\mathcal{P}$ is realizable by web service composition $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$ if the lossless join, synchronous compatible and autonomous conditions are satisfied.

*Proof:* Convert FC conversation protocol $\mathcal{P}$ to the equivalent FSA conversation protocol $\mathcal{P}'$ as shown in the proof of Theorem 4.2. Based on [9], when the three realizability conditions are satisfied, $\mathcal{P}'$ is realized by its projected composition, which can be converted to $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$. ■

Unfortunately, we cannot state similar results for the IC conversation protocols. We suspect that even if an IC conversation protocol satisfies these three realizability conditions (where the decision problem for these three conditions may not even be decidable), there may not exist a finite control state system which realizes the protocol. In the following sections we will describe other analysis techniques based on the realizability conditions listed above which can be applied to both FC and IC conversation protocols.

## VI. SKELETON ANALYSIS

This section investigates the feasibility of deciding if a conversation protocol is realizable by checking its abstract control flows (called skeletons), without considering its data semantics. Note that the skeleton analysis works for both FC and IC conversation protocols.

*Definition 6.1:* Given a GA $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$, its skeleton, denoted as $\mathrm{skeleton}(\mathcal{A})$, is a standard FSA $(M, T, s, F, \delta')$ where $\delta'$ is obtained from $\delta$ by replacing each transition $(s, (c, g), t)$ with $(s, c, t)$.

Note that $L(\mathrm{skeleton}(\mathcal{A})) \subseteq M^*$, while $L(\mathcal{A})$ is a subset of $\Sigma^*$. For a conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, we can always construct an FSA conversation protocol $\langle (P, M), \mathrm{skeleton}(\mathcal{A}) \rangle$. We call this protocol the *skeleton protocol* of $\mathcal{P}$.

Now, one natural question is the following:

*If the skeleton protocol of a conversation protocol is realizable, does this imply that the original protocol is realizable?*

The following is a counter example against the above conjecture.

*Example 6.1:* The conversation protocol shown in Fig. 8 has four peers $A, B, C, D$. There are two message classes in the system: $\alpha$ from $A$ to $B$ and $\beta$ from $C$ to $D$. Both message classes have an attribute $\mathtt{a}$. The protocol

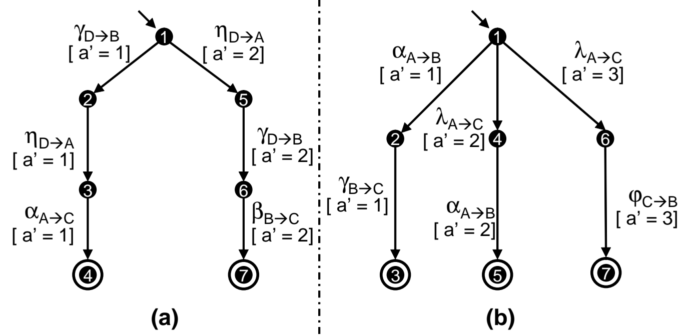


Fig. 9. The Conversation Protocols for Examples 6.2 and 6.3

specifies two possible conversations $\alpha(1)\beta(1)$, and $\beta(2)\alpha(2)$. Obviously, the skeleton protocol, which specifies the desired conversation set $\{\alpha\beta, \beta\alpha\}$, is realizable because it satisfies all the three realizability conditions. However, the original protocol itself is not realizable, because any implementation that generates the specified conversations will also generate the conversation $\beta(1)\alpha(1)$. ∎

*Example 6.2:* Note that the skeleton of the example shown in Fig. 8 is lossless join, however the conversation protocol itself is not. Fig. 9(a) shows an example where the protocol is lossless join, while its skeleton is not. There are four peers $A, B, C, D$ in Fig. 9(a), and all message classes contain 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 $B$ know who is going to send the last message ($\alpha$ or $\beta$), so there is no ambiguity. It can be verified that the protocol is lossless join. However the skeleton of Fig. 9(a) is obviously not lossless join, because $\eta\gamma\alpha$ is included in its join closure. ∎

*Example 6.3:* If we make the message $\beta$ in Fig. 8 from peer $C$ to $A$, then the modified protocol is an example which is not synchronous compatible, yet its skeleton is synchronous compatible. Fig. 9(b) shows another example, where the conversation protocol itself is synchronous compatible however its skeleton is not, because after the partial conversation $\lambda\alpha$, peer $B$ is ready to send $\gamma$ however peer $C$ is not receptive to it. ∎

The following propositions summarize the observations from Examples 6.1, 6.2, and 6.3.

*Proposition 6.4:* A conversation protocol may be realizable (lossless join, synchronous compatible, autonomous) while its skeleton protocol is not realizable (lossless join, synchronous compatible, autonomous).

*Proposition 6.5:* A conversation protocol may not be realizable (lossless join, synchronous compatible, autonomous) when its skeleton protocol is realizable (lossless join, synchronous compatible, autonomous).

The above propositions suggest that we cannot tell if a conversation protocol is realizable or not, based on the properties of its skeleton protocol. However, in the following we will show that with an additional condition we can use the properties of a protocol's skeleton to reason about its realizability.

*A. Skeleton Analysis*

We now introduce a fourth realizability condition to restrict a conversation protocol so that it can be realized by $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$, $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$, and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$ when its skeleton satisfies the three realizability conditions discussed above.

*Definition 6.2:* Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ be a conversation protocol where $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$. $\mathcal{P}$ is said to satisfy the *deterministic guards condition* if for each pair of transitions $(t_1, (m_1, g_1), t_1')$ and $(t_2, (m_2, g_2), t_2')$, $g_1$ must be equivalent to $g_2$, when the following conditions hold:

1) $m_1 = m_2$, and
2) Let $p_i$ be the sender of $m_1$. There exists two words $w \in L(\mathcal{A}^*)$ and $w' \in L(\mathcal{A}^*)$ where a partial run of $w$ reaches $t_1$, and a partial run of $w'$ reaches $t_2$, and $\pi_i(\pi_{\mathrm{TYPE}}(w)) = \pi_i(\pi_{\mathrm{TYPE}}(w'))$. Here $\pi_{\mathrm{TYPE}}$ of a conversation casts the conversation to a sequence of message classes, which one to one corresponds to the messages in the conversation.

Intuitively, the deterministic guards condition requires that for each peer, according to the conversation protocol, when it is about to send out a message, the guard that is used to compute the contents of the message is uniquely decided by the sequence of message classes (note, not message contents) exchanged by the peer in the past.

The decision procedure for the deterministic guards condition proceeds as follows: given a conversation protocol $\mathcal{P}$, obtain its coarse-1 projected composition $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$. Let $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. For each $i \in [1..n]$, regard $\mathcal{A}_i$ as a standard FSA, and get its equivalent deterministic FSA (let it be $\mathcal{A}_i'$). Now each state $t$ in $\mathcal{A}_i'$ corresponds to a set of states in $\mathcal{A}_i$, and let it be represented by $T(t)$. We examine each state $t$ in $\mathcal{A}_i'$, for each message class $c \in M$, we collect the guards of the transitions that start from a state in $T(t)$ and send message with class $c$. We require that all guards collected for a state/message class pair ($t$,$c$) should be equivalent.

*Example 6.6:* The conversation protocol in Fig. 8 violates the deterministic guards condition, because (intuitively) peer $A$ has two different guards when sending out $\alpha$ at the initial stage. Formally, to show that the deterministic guards condition is violated, we can find two transitions $(t_1, (\alpha, [a' = 1]), t_2)$ and $(t_3, (\alpha, [a' = 2]), t_4))$, and two words $w = \epsilon$ and $w' = \beta(2)$. Since a run of $w$ reaches $t_1$, a run of $w'$ reaches $t_3$, and $\pi_A(\pi_{\mathrm{TYPE}}(w)) = \pi_A(\pi_{\mathrm{TYPE}}(w')) = \epsilon$, by Definition 6.2, the guards of the two transitions should be equivalent. However, they are not equivalent, and this leads to the violation of the deterministic guards condition. ∎

*Theorem 6.7:* A conversation protocol $\mathcal{P}$ is realized by $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$, $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$, and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$ if it satisfies the deterministic guards condition, and its skeleton protocol satisfies the lossless join, synchronous compatibility and autonomy conditions.

*Proof:* Let $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ},\mathbb{C}1}$ be the web service composition generated from $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ by determinizing each peer and collecting guards as in the check of the deterministic guards condition. We show that $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ},\mathbb{C}1})$. Other results such as $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2})$ and $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}})$ can be directly inferred from Lemma 4.4 and the $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ},\mathbb{C}1})$ we are about to prove.

First, we argue that if the skeleton protocol satisfies the synchronous compatibility and autonomy conditions, then during any (complete or partial) run of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ},\mathbb{C}1}$, each message is consumed "eagerly", i.e., when the input

queue is not empty, a peer never sends out a message or terminates. This argument can be proved by contradiction. Assume there is a partial run against this argument , i.e., we can find a corresponding partial run of the skeleton composition of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}$ (which consists of the skeletons of each peer of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}$) where a message class is not consumed eagerly (without loss of generality, suppose this is the shortest one). Then there must be a pair of consecutive configurations where a peer $i$ has a message at the head of its queue and it sends a message rather than receiving the message. Due to synchronous compatible condition we know that the peer $i$ should be able to receive the message at the head of the queue immediately after it was sent. We also know that due to autonomy condition peer $i$ can only execute receive transitions if it has one receive transition from a configuration. Then, sending a message will violate these conditions and create a contradiction. Hence we conclude that each message class is consumed eagerly.

Now it suffices to show that $\mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}) \subseteq L(\mathcal{P})$, as $L(\mathcal{P}) \subseteq \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}})$ is obvious. Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A}\rangle$ and let $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$. Given a word $w \in \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}})$, and $\gamma$ be the corresponding run, we can always construct a run $\gamma'$ of $\mathcal{A}$ to recognize $w$. Since $\pi_i(w)$ is accepted by each peer $\mathcal{A}_i$, $\pi_i(\pi_{\mathrm{TYPE}}(w))$ is accepted by $\mathrm{skeleton}(\mathcal{A}_i)$. Because $\mathrm{skeleton}(\mathcal{A})$ is lossless join, it follows that $\pi_{\mathrm{TYPE}}(w)$ is accepted by $\mathrm{skeleton}(\mathcal{A})$, and let $\mathcal{T} : \tau_1\tau_2 \ldots \tau_{|w|}$ be the path of $\mathrm{skeleton}(\mathcal{A})$ traversed to accept $\pi_{\mathrm{TYPE}}(w)$. Since each transition in $\mathrm{skeleton}(\mathcal{A})$ is the result of dropping the guard of a corresponding transition, we can have a corresponding path $\mathcal{T}'$ in $\mathcal{A}$ by restoring message contents. Notice that we can always do so because in each step, the global configuration allows the guards to be evaluated as if it is executed synchronously. This results from the fact that whenever a message is to be sent, its contents always depend on the latest copies of arrived messages, because queue is empty, and every input message which causally happens before it has already been consumed.

■

Based on Theorem 6.7, we obtain a light-weight realizability analysis for 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 checking the guards (but actually without analyzing their data semantics).

## VII. Symbolic Analysis

Sometimes skeleton analysis may be too coarse and fail to show the realizability of a realizable conversation protocol. For example, Fig. 10(a) is a realizable alternating bit protocol, however, the skeleton analysis fails to show that it is realizable. The conversation protocol shown in Fig. 10(a) consists of two peers $A$ and $B$. Message class $\alpha$ is a request, and message class $\beta$ is an acknowledgment. Both message classes contain an "`id`" attribute. Message class $\gamma$ is the end-conversation notification. The protocol states that the `id` attribute of requests from $A$ alternates between 0 and 1, and every acknowledgment $\beta$ must match the `id`.

Let $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ be the three conversation protocols shown in Fig. 10. 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

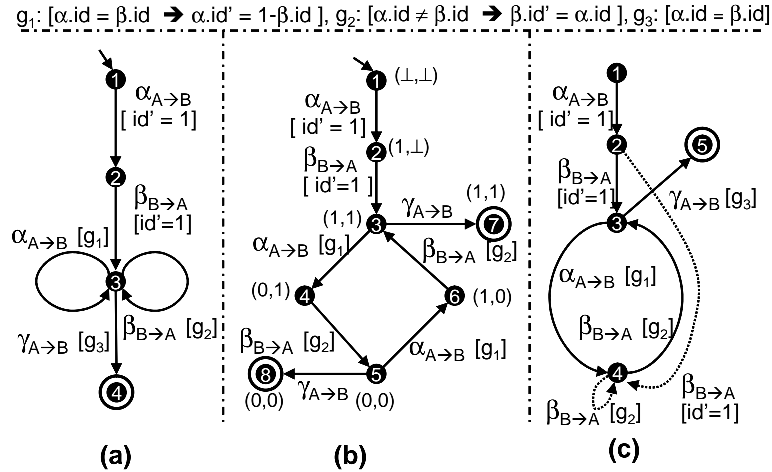


Fig. 10. Alternating Bit Protocol

"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 $\alpha$ and $\beta$. 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 $L(\mathcal{A}_a) = L(\mathcal{A}_b) = L(\mathcal{A}_c)$.

### *A. Analysis of Autonomy Using Iterative Refinement*

The examples in Fig. 10 motivates analyzing the autonomous condition using iterative refinement: 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 $\mathcal{A}_a$). In the following, we first present the theoretical background for the analysis of the autonomy condition using iterative refinement.

This analysis is based on the notion of *simulation*, which is defined below. A transition system is a tuple $(M, T, s, \Delta)$ where $M$ is the set of labels, $T$ is the set of states, $s$ the initial state, and $\Delta$ the transition relation. Generally a transition system can be regarded as an FSA (or an infinite state system) without final states. On the other hand, a standard FSA $(M, T, s, F, \Delta)$ can be regarded as a transition system of $(M, T, s, \Delta)$; and a GA $(M, \Sigma, T, s, F, \Delta)$ can be regarded as a transition system of the form $(\Sigma, T', s', \Delta')$ where $T'$ contains all configurations of the GA, and $\Delta'$ defines the derivation relation between configurations.

*Definition 7.1:* 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 $\rho : T \to T'$ and $\varrho : M \to M'$ such that for each $(s, m, t)$ in $\Delta$ there is a $(\rho(s), \varrho(m), \rho(t))$ in $\Delta'$. 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 $\mathcal{A}' \preceq \mathcal{A}$.

**Procedure** AnalyzeAutonomy($\mathcal{A}$): List
**Begin**
$\mathcal{A}'$ = DeterminizeGA($\mathcal{A}$)
**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**

Fig. 11. Iterative Analysis

*Example 7.1:* For the three conversation protocols $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ in Fig. 10, the following is true:

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

For example, in the simulation relation $\text{skeleton}(\mathcal{A}_c) \preceq \text{skeleton}(\mathcal{A}_a)$, $\rho$ 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 $\varrho$ is the identity function which maps each message class to itself. For another example, $\mathcal{A}_a \preceq \text{skeleton}(\mathcal{A}_a)$, and $\mathcal{A}_a \simeq \mathcal{A}_b \simeq \mathcal{A}_c$. ∎

Intuitively when $\mathcal{A} \preceq \mathcal{A}'$, each word accepted by $\mathcal{A}$ has 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 7.2:* For any GA $\mathcal{A}$, $\mathcal{A} \preceq \text{skeleton}(\mathcal{A})$.

*Proof:* We can construct the mappings from $\mathcal{A}$ to its skeleton. Since a configuration of $\mathcal{A}$ 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 configuration $c = (t, \vec{m})$, $\rho(c) = t$, and for each message $m$, $\varrho(m) = \text{TYPE}(m)$ . ■

*Lemma 7.3:* For each GA $\mathcal{A} = (M, \Sigma, T, s, F, \Delta)$ on a finite alphabet $\Sigma$, there is a standard FSA on alphabet $\Sigma$ such that $\mathcal{A} \simeq \mathcal{A}'$.

*Proof:* This lemma directly follows Theorem 4.2. ■

*Theorem 7.4:* If $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{A}'$ is autonomous, then $\mathcal{A}$ is autonomous.

*Proof:* The proof follows directly from the fact that each run of $\mathcal{A}$ has a corresponding run in $\mathcal{A}'$. If during each run of $\mathcal{A}'$, autonomy condition is not violated, obviously any run of $\mathcal{A}$ will not violate it either. ■

Lemma 7.2 and Theorem 7.4 immediately leads to the following.

*Corollary 7.5:* A GA is autonomous if its skeleton is autonomous.

Based on Corollary 7.5 we have an error-trace guided symbolic analysis algorithm (procedure AnalyzeAutonomy in Fig. 11). If the input GA is autonomous, AnalyzeAutonomy returns `null`; otherwise it returns the error trace which is a list of configurations. AnalyzeAutonomy starts from the input GA, and refines incrementally. During each cycle, procedure AnalyzeAutonomy analyzes the skeleton of the current GA $\mathcal{A}'$. If the skeleton is autonomous, by Corollary 7.5, AnalyzeAutonomy simply returns and reports that the input GA is autonomous; otherwise,

**Procedure** Refine($\mathcal{A}$, $(s, (m_1, g_1), t_1)$, $(s, (m_2, g_2), t_2)$) : (GA, List)
**Begin**
  **If** $(\textsf{Pre}(g_1) \wedge \textsf{Pre}(g_2))$ is satisfiable **then**
    Path = FindPath($\mathcal{A}$, $s$, $\textsf{Pre}(g_1) \wedge \textsf{Pre}(g_2)$)
    **If** Path $\neq$ null **then** return (null,Path)
  **End If**
  **Let** $\mathcal{A}' = (M', T', s_0', F', \Delta')$ be a copy of $\mathcal{A}$
  $T' = T' - \{s\} + \{s_1, s_2\}$, $F' = F' - \{s\} + \{s_1, s_2\}$ if $s \in F'$
  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 $(s_2, (m_2, g_2), t_2)$
  Remove all unreachable transitions
  return ($\mathcal{A}'$,null)
**End**

Fig. 12. Refinement of Guarded Automata

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 Fig. 10(a), the two transitions starting at state 3 will be identified. Then procedure Refine is invoked to refine the current GA. This refinement process continues until the input GA is proved to be autonomous or an concrete error trace is found.

*1) Refining a Guarded Automaton:* We present the algorithm to refine a Guarded Automaton in Fig. 12. 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 GA by splitting the source state of these two transitions. If refinement succeeds, the refined GA is returned; otherwise, a concrete error trace is returned to show that the input GA is not autonomous.

The first step of Refine is to compute the conjunction of the precondition of the two guards, i.e., $\textsf{Pre}(g_1) \wedge \textsf{Pre}(g_2)$. 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 GA.

*Example 7.6:* When procedure Refine is applied to Fig. 10(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 Fig. 10(c)), and transitions are modified accordingly. Finally, unreachable transitions (dotted arrows in Fig. 10(c)) are removed, and we get the GA in Fig. 10(c). ∎

The precondition operator Pre is a standard operator in symbolic model checking, in which, all primed variables are eliminated using existential quantifier elimination. For example given a constraint $g$ as "$a = 1 \wedge b' = 1$", its

precondition is $\mathsf{Pre}(g) = \exists_{a'}\exists_{b'}(a = 1 \wedge b' = 1)$, which is equivalent to "$a = 1$".

*2) Generating a Concrete Error Trace:* We present the algorithm to locate a concrete error trace in Fig. 13. Procedure $\mathrm{FindPath}$ has three inputs: a GA $\mathcal{A}$, 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 $g$.

**Procedure** FindPath($\mathcal{A}$, $s$, $g$): List
**Begin**
  **Let** $\mathcal{A} = (M, T, s_0, F, \Delta)$
  **Let** $\mathcal{T}$ be $\bigcup_{(s_i,(m_j,g_k),s_\ell)\in\Delta} \; g_k \wedge state = s_i \wedge state' = s_\ell$
  Stack path = new Stack()
  **Let** $g$ be re-assigned as $g \wedge state = s$
  $g'$ = `false`
  stack.push($g$)
  **While** $g \neq g'$ and $g \wedge state = s_0$ is not satisfiable **do**
    $g'$ = $g$
    $g$ = $(\exists_{M'} \; (g_{M/M'} \wedge \mathcal{T})) \vee g'$
    path.push($g$)
  **End While**
  **If** $g \wedge state = s_0$ is not satisfiable **Then**
    return null
  **Else**
    path = reverse order of path
    List ret = new List()
    cvalue = a *concrete value* of path[1]
    **For** $i$ =1 to $|path|$ **do**
      ret.append(cvalue)
      cvalue = a concrete value in $(\exists_M \; \mathrm{cvalue} \wedge \mathcal{T})_{M'/M}$
    **End For**
    return ret
  **End If**
**End**

Fig. 13. Generation of a Concrete Error Trace

The algorithm of $\mathrm{FindPath}$ is a variation of the standard symbolic backward reachability analysis in model checking techniques. The procedure starts with the construction of a symbolic transition system $\mathcal{T}$, based on the control flow as well as data semantics of $\mathcal{A}$. Then given the initial constraint $g$, the main loop computes the constraint which generates $g$ via transition system $\mathcal{T}$. The loop terminates when it reaches the initial configuration, or it reaches a fixed point.

*Example 7.7:* If we redefine the guard $g_2$ as $\alpha.id = \beta.id \Rightarrow \beta.id' = \alpha.id$. When procedure $\mathrm{Refine}$ is called on Fig. 10(a), the conjunction of preconditions of $g_1$ and $g_2$, i.e., $\alpha.id = \beta.id$, is satisfiable. Hence procedure $\mathrm{FindPath}$ is called with input Fig. 10(a), state 3, and constraint $\alpha.id = \beta.id \wedge state = s3$. The while loop of $\mathrm{FindPath}$ eventually includes in variable $path$ the following constraints:

1) $\alpha.id = \beta.id \wedge state = s3$.
2) $\alpha.id = 1 \wedge state = s2$.

3) $state = s1$.

Then the order of $path$ is reversed, and a concrete value constraint $cvalue$ is randomly generated which satisfies constraint $state = s1$ and each message attribute has an exact value in $cvalue$, for example, let $cvalue$ be $\alpha.id = 1 \wedge \beta.id = 0$, then the list $ret$ will record the following constraints:

1) $\alpha.id = 1 \wedge \beta.id = 0 \wedge state = s1$.
2) $\alpha.id = 1 \wedge \beta.id = 0 \wedge state = s2$.
3) $\alpha.id = 1 \wedge \beta.id = 1 \wedge state = s3$.

Obviously the sequence of constraints in $ret$ captures an error trace leading to the state 3 which violates the autonomy condition. ∎

Complexity of the algorithms in Fig. 11, Fig. 12, and Fig. 13 depends on the data domains associated with the input GA. When the message alphabet of a guarded conversation protocol is finite, algorithms in Fig. 12 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.

### *B. Symbolic Analysis of Synchronous Compatibility*

One natural question is: do we have similar iterative analysis algorithms for the lossless join and synchronous compatibility conditions? The answer is negative, because of Propositions 6.4 and 6.5. However, we do have "conservative" symbolic analyses for these two conditions. In the following, we first discuss the symbolic analysis for synchronous compatibility.

Recall the algorithm to check synchronous compatibility of an FSA conversation protocol. We project the protocol to each peer and determinize them (including $\epsilon$-transition elimination), 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 analysis of synchronous compatibility for a GA conversation protocol follows exactly the same procedure. But note that, we have to discuss two different cases on GA conversation protocols with finite or infinite domains. Given an FC conversation protocol $\mathcal{P}$, we can always construct its exact equivalent FSA conversation protocol (let it be $\mathcal{P}'$), and use the synchronous compatibility analysis for standard FSA protocols to analyze $\mathcal{P}'$. However, for IC conversation protocols we might not be able to do so, because there may not exist projections for IC conversation protocols. In the following, we introduce a "conservative" symbolic analysis for the synchronous compatible condition.

Given an IC (or FC) conversation protocol $\mathcal{P}$, we can project it to each peer using coarse projection (either Coarse Processing 1 or Coarse Processing 2 in Fig. 5). Then we determinize each peer in $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ (or $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$) using the DeterminizeGA in Fig. 7. We construct the Cartesian product of those determinized GA using the algorithm presented in Section 4. If no illegal state is found, the IC conversation protocol $\mathcal{P}$ is synchronous compatible; however, this method is conservative, i.e., if an illegal state is found, $\mathcal{P}$ might still be synchronous compatible, because a coarse projection accepts a superset of the language accepted by the exact projection.

### C. Symbolic Analysis of Lossless Join

Similar to the analysis of synchronous compatibility, for an FC conversation protocol, we can always construct its equivalent FSA conversation protocol and apply the lossless join check for FSA conversation protocols. Now we discuss a conservative and symbolic analysis for IC as well as FC conversation protocols.

Recall that each GA $\mathcal{A}$ can be regarded as a transition system, and can be represented symbolically. Let $\mathcal{T}(\mathcal{A})$ denote the symbolic transition system derived from $\mathcal{A}$. From the initial configuration of $\mathcal{A}$, we can compute all the reachable configurations of $\mathcal{T}(\mathcal{A})$, and let set of reachable configurations be $S^{\mathcal{A}}$. Given $\mathcal{A}_1$ and $\mathcal{A}_2$, the following statement is true:

$$(S^{\mathcal{A}_1} \wedge \mathcal{T}(\mathcal{A}_1) \;\Rightarrow\; S^{\mathcal{A}_2} \wedge \mathcal{T}(\mathcal{A}_2)) \;\Rightarrow\; (L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)) \tag{1}$$

Intuitively, Equation 1 means that if $\mathcal{A}_2$ as a transition system is a superset of $\mathcal{A}_1$, i.e., for any reachable configuration, there are more enabled transitions in $\mathcal{T}(\mathcal{A}_2)$ than $\mathcal{T}(\mathcal{A}_1)$, then $L(\mathcal{A}_2)$ should be a superset of $L(\mathcal{A}_1)$.

Equation 1 naturally implies the symbolic analysis algorithm. Given a conversation protocol $\mathcal{P}$ (with finite or infinite domains), let its GA specification be $\mathcal{A}$. We can project $\mathcal{A}$ using coarse projection. Then construct the Cartesian product of $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ (or $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$), and let it be $\mathcal{A}'$. Then we construct $\mathcal{T}(\mathcal{A})$, $\mathcal{T}(\mathcal{A}')$, and compute $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$. Finally if $(S^{\mathcal{A}'} \wedge \mathcal{T}(\mathcal{A}')) \;\Rightarrow\; (S^{\mathcal{A}} \wedge \mathcal{T}(\mathcal{A}))$, we can conclude that $\mathcal{P}$ is lossless join.

The above symbolic analysis algorithm is decidable when the domain is finite. When $\mathcal{P}$ has an infinite domain, we can simply use the approximate closure of $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$, and it is still a conservative algorithm.

### D. Hybrid Analyses

Up to now, we have introduced a range of techniques to analyze each of the realizability conditions. The most light-weight analysis is the skeleton analysis. However, this approach may not be very precise. One possibility is to use symbolic analyses to improve skeleton analysis. Given a GA-conversation protocol $\mathcal{P}$, we can first check the autonomy condition using the iterative refinement algorithm. If $\mathcal{P}$ is proved to be autonomous, we can apply the skeleton analysis on the refined protocol obtained in the iterative analysis. When the skeleton analysis succeeds, $\mathcal{P}$ is guaranteed to be realized by $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$, $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$, and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$.

## VIII. Conclusions

In this paper we presented results on the realizability analysis for conversation protocols with message contents. We presented a guarded automata model for specifying conversation protocols with message contents. The realizability analysis in the guarded automata model is more complex than the realizability analysis of conversation protocols without message contents. We showed that, the realizability of the "skeleton" of a conversation protocol does not imply the realizability of the conversation protocol itself. Only by enforcing an additional condition, we are able to identify some classes of realizable conversation protocols. In addition, the size of the domain of message classes can significantly affect the efficiency and the theoretical characteristics of the realizability analysis. Finite domain

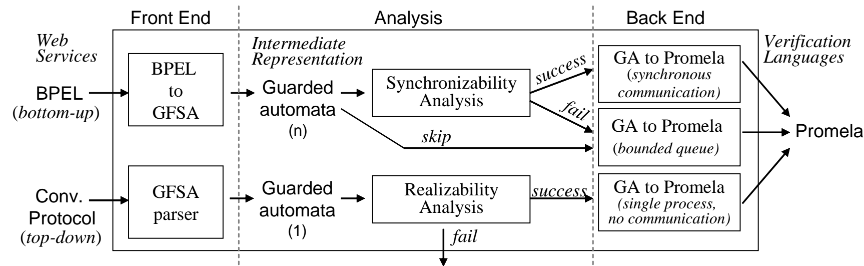


Fig. 14. WSAT architecture

conversation protocols are shown to have exact projections, however, the same argument does not hold for infinite domain conversation protocols. We showed that skeleton analysis may not be precise enough—some realizable conversation protocols cannot be shown to be realizable with this analysis. We proposed the refined symbolic realizability analysis for all realizability conditions, which can improve both the accuracy and the efficiency of the analysis.

The skeleton realizability analysis presented in this paper has been implemented as a part of the Web Service Analysis Tool (WSAT) [12]. In Fig. 14, we present the general architecture of WSAT. The front-end of WSAT accepts industry web service standards such as WSDL [22] and BPEL [2]. The core analysis engine of WSAT is based on the intermediate representation GA. The back-end employs model checker SPIN [15] for verification. At the front-end, a translation algorithm from BPEL4WS to GA is implemented, and support for other languages can be added without changing the analysis and the verification modules of the tool. At the core analysis part, realizability analysis and another similar analysis called "synchronizability analysis" are implemented to avoid the undecidability caused by asynchronous communication. At the back-end, translation algorithms are implemented from GA to Promela, the input language of SPIN. Based on the results of the realizability and the synchronizability analyses, the LTL verification at the back-end can be performed using the synchronous communication semantics instead of asynchronous communication semantics.

We applied WSAT to a range of examples, including six conversation protocols converted from the IBM Conversation Support Project [16], five BPEL4WS services from BPEL4WS standard and Collaxa.com, and the SAS from [10]. We applied the synchronizability and the realizability analyses to these examples, and except for two conversation protocols, all examples were shown to be either realizable or synchronizable. This implies that the sufficient conditions in our realizability analysis are not restrictive and they are able to capture most practical applications.

## ACKNOWLEDGMENTS

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

# REFERENCES


[1] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S Williams. Web Services Conversation Language (WSCL) 1.0. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/, March 2002.

[2] Business process execution language for web services (BPEL), version 1.1. *available at* http://www.ibm.com/developerworks/library/ws-bpel.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

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

[5] DAML-S (and OWL-S) 0.9 Draft Release. http://www.daml.org/services/daml-s/0.9/, May 2003.

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003)*, 2003.

[7] 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 2003)*, volume 2759 of *LNCS*, pages 188–200, 2003.

[8] X. Fu, T. Bultan, and J. Su. Analysis of interacting web services. In *Proceedings of the Twelfth International World Wide Web Conference (WWW 2004)*, pages 621 – 630, New York, May 2004.

[9] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. to appear in *Theoretical Computer Science*, 2004.

[10] X. Fu, T. Bultan, and J. Su. Model checking xml manipulating software. To appear in the *2004 Int. Symp. on Software Testing and Analysis (ISSTA)* , 2004.

[11] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. In *Proc. of 2004 IEEE Int. Conf. on Web Services (ICWS)*, pages 96–103, San Diego, CA, July 2004.

[12] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web service compositions. In *Proc. of 16th Int. Conf. on Computer Aided Verification (CAV)*, 2004.

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

[14] J. E. Hanson, P. Nandi, and D. W. Levine. Conversation-enabled web services for agents and e-business. In *Proc. Int. Conf. on Internet Computing (IC-02)*, pages 791–796. CSREA Press, 2002.

[15] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] IBM. Conversation Support Project. http://www.research.ibm.com/convsupport/.

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

[18] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.

[19] MicroSoft Message Queuing Service. http://www.microsoft.com/msmq/.

[20] 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.

[21] Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/2002/NOTE-wsci-20020808/, August 2002.

[22] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.