Running head: REALIZABILITY ANALYSIS

Realizability Analysis of Top-down Web Service Composition Specifications

Xiang Fu

School of Computer and Information Sciences, Georgia Southwestern State University

Tevfik Bultan

Department of Computer Science, University of California, Santa Barbara

Jianwen Su

Department of Computer Science, University of California, Santa Barbara

Abstract

A conversation protocol specifies the desired global behaviors of a web service composition in a top-down fashion. Before implementing a conversation protocol, its realizability has to be determined, i.e., can a bottom-up web service composition be synthesized so that it generates exactly the same set of conversations as specified by the protocol? This chapter presents three sufficient conditions to restrict control flows of a conversation protocol for achieving realizability. The model is further extended to include data semantics of web services into consideration. To overcome the state-space explosion problem, symbolic analysis techniques are used for improving the accuracy of analysis. The realizability analysis can effectively reduce the complexity of verifying web services with asynchronous communication.

KEY WORDS:

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

Realizability Analysis of Top-down Web Service Composition Specifications

To construct a mission critical web service composition (also called "composite web service") is a very challenging task, as any design or implementation fault could lead to great losses. Recently, automated verification and testing of web services have attracted attention in both academia and industry (Bultan, Fu, Hull, & Su, 2003; Foster, Uchitel, Magee, & Kramer, 2003; Narayanan, & Mcllraith, 2003; Betin-Can, Bultan, & Fu, 2005; Canfora & Di Penta, 2006). However, before any automatic verification technique can be applied, a formal model has to be defined to describe behaviors of web services. This chapter presents a top-down specification approach called "conversation protocol" and studies the realizability problem of conversation protocols. It is an extension of the work by Fu, Bultan, and Su (2004c, 2005b) and covers other results by Fu et al. (2003, 2004a, 2004b, 2004d, 2004e, 2005a) in the area.

INTRODUCTION

In general, there are two different ways of specifying a **web service** composition: (1) The bottom-up approach, favored by many industry standards such as WSDL (World Wide Web Consortium [W3C], 2001) 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., Message Sequence Charts (ITU-T , 1994), conversation policies (Hanson, Nandi, & Kumaran, 2002), WSCI (W3C, 2002), and WSCL (Banerji, Bartolini, Beringer, Chopella, Govindarajan, Karp, et al., 2002) in which the set of desired message exchange patterns is specified first and detailed specification of peer implementation is left blank.

In this chapter we concentrate on the top-down specification approach due to its simplicity and the potential benefits in verification complexity (Bultan, Fu, Hull, & Su, 2003). One natural idea for top-down specification of web services is to use finite state machines (FSA) to represent

some aspects of the global composition process. The state machines can involve two parties (Hanson, Nandi, & Levine, 2003) or multi-parties (Bultan et al., 2003), and may describe the global composition process directly (Hanson, et al., 2003) or may specify its local views (Banerji, Bartolini, Beringer, Chopella, Govindarajan, Karp, et al., 2002).

A top-down conversation protocol has to be realized by a bottom-up web service composition. In studying the composition behaviors, asynchrony usually complicates analyses. Asynchronous communication is one of the benefits provided by the web service technique. It is supported by many industry platforms such as Java Message service (Sun, n.d.) and Microsoft Message Queuing service (Microsoft, n.d.). In an asynchronous communication environment, receiver of a message does not have to synchronize its action with the send action by the sender. However, asynchrony may significantly increase the complexity of many verification problems. Fu, Bultan, and Su (2003) proved that the general problem of verifying a Linear Temporal Logic property on a bottom-up specified web service composition is undecidable, which is essentially caused by the undecidable nature of communicating finite state machines (Brand & Zafiropulo, 1983).

Asynchronous communication is usually modeled by equipping participating services with FIFO queues. For example, Bultan, Fu, Hull, and Su (2003) established a conversation oriented framework where each participating web service (called a “peer”) of a composition is characterized using a finite state automaton, with the set of input/output message classes 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 (Clarke, Grumberg, & Peled, 2000) can be naturally extended to this

conversation based framework. Desired system properties such as "a cancel request always results in a confirmation message" can be expressed using Linear Temporal Logic and the web service composition can then be verified using automatic verification tools such as the Web Service Analysis Tool (Fu, Bultan, & Su, 2004d).

A conversation protocol is not always realizable, i.e., there exists conversation protocols which do not have any peer implementations whose composition generates exactly the same set of conversations as specified by the protocol. Hence, before implementing a conversation protocol, its realizability has to be studied first. Fu, Bultan, and Su (2003) proposed three sufficient conditions that can guarantee the realizability of conversation protocols. Later the realizability analysis technique is extended to a model with data semantics (Fu et al, 2004c).

Related Work

Realizability of software systems has been investigated for decades in different branches of computer science. In the late 1980's, researchers proposed the realizability problem of open systems (Abadi, Lamport, & Wolper, 1989; Pnueli, & Rosner, 1989). It studies whether a peer has a strategy to cope with the environment no matter how environment moves. A closer notion to the realizability problem studied in this chapter is the concept of "weak/strong realizability" on the Message Sequence Chart Graphs (MSCG) model by Alur, McMillan, and Peled (2000) and Alur, Etessami, and Yannakakis (2001). In the MSCG model each peer is also equipped with a message buffer to simulate the asynchronous communication environment. The difference is that the MSCG model includes both send and receives events, while the conversation model used in this chapter studies the send events only. This leads to delicate difference in the analysis complexity and the realizability conditions on the two models. A detailed comparison of the two

models can be found in Fu, Bultan, and Su's work in IEEE Transaction on Software Engineering (2005b).

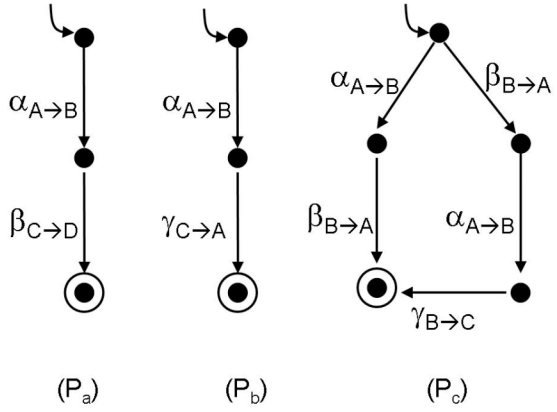

Figure 1. Three non-realizable conversation protocols, adapted from the Büchi autumata protocol examples by Fu, Bultan, and Su (2003).

BASIC REALIZABILITY ANALYSIS WITHOUT MESSAGE CONTENTS

Consider the three conversation protocols presented in Figure 1, and let them be P_a, P_b and P_c. Each conversation protocol is expressed using a finite state machine. In a finite state machine, the initial state is denoted using an incoming edge without source, and a final state is denoted using a double cycle. Each of these three protocols involves four peers *A*, *B*, *C*, and *D*.

Conversation protocol P_a is not realizable. It specifies one single conversations αβ, where message α is sent from *A* to *B* and message β is sent from *C* to *D*. Clearly any implementation of the four peers which generates a conversation αβ can also generate the conversation βα. Similarly P_b is not realizable because there is no way for peer *C* to learn about the time message α is sent. Hence, any peer implementation which generates αγ will also generate γα.

Protocol P_c defines a conversation set: $\{\alpha\beta, \beta\alpha\gamma\}$. This protocol is more interesting in the sense that peers might have "confusion" when executing the protocol. It is possible that peers *A* and *B* take the left and the right branches of the protocol (respectively), unaware that the other peer is taking a different branch. For example, the following execution sequence can generate a conversation not specified by the protocol although each peer is executing the protocol faithfully. Peer *A* first sends a message α, which is then stored in the queue of *B*; then *B* sends a message β, which is then 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 γ. Hence a conversation αβγ is produced by the peer implementations. However, this conversation is not specified by P_c.

Fu, Bultan, and Su (2003) proposed three realizability conditions for conversation protocols (without message contents). When these sufficient conditions are satisfied, a conversation protocol is guaranteed to be realizable. In the following we briefly present these conditions.

(1) Lossless join condition: A conversation protocol is said to be lossless join if its conversation set is equal to the join of its projections to all peers. Here, the terms "join" and "projection" are borrowed from relational database theory.

The projection of a conversation to a peer *i* generates a message sequence by removing all the messages that are not related to peer *i* from the given conversation. For example, consider the conversation αβ defined by P_a in Figure 1. Its projection to peer *A* is α (where message β is removed from the conversation because *A* is neither its sender nor its receiver); similarly, the projection of αβ to *C* is β.

Given a set of languages (one language for each peer), their join is a set which includes all conversations whose projection to each peer is included in the corresponding language for that peer. For example, suppose message α is sent from *A* to *B* and message β is sent from *C* to *D*.

The join of four languages {α}, {α}, {β}, {β} (for *A*, *B*, *C*, and *D* respectively) results in the set {αβ, βα}.

A conversation protocol is said to be a lossless join if its conversation set is the join of its projections to all peers. For example, the join of the projections of P_a to all peers is {αβ, βα} which is a strict superset of the conversation set specified by P_a. Hence, P_a is not lossless join. However, it can be verified that P_b and P_c in Figure 1 satisfy the lossless join condition.

Lossless join condition is actually a necessary condition of realizability. It is not hard to see that it reflects the requirement that a conversation protocol should be complete in the sense that the relative order of messages should not be restricted if they do not have causal relationship.

(2) Synchronous compatibility condition: Intuitively a conversation protocol is synchronous compatible if its straightforward implementation (by projecting the protocol to each peer) can work without conflicts using synchronous communication semantics. Formally, 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.

The synchronous compatibility condition is checked as follows: (a) Project the protocol to each peer by replacing all the transitions labeled with messages for which that peer is neither the sender nor the receiver with ε-transitions (empty moves); (2) Determinize the resulting automaton for each peer; (3) Generate a product automaton by combining the determinized projections based on the synchronous communication semantics (i.e., each pair of send and receive operations are taken simultaneously at the sender and receiver); (4) 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 Figure 1. Its projection to peer *A* returns the same FSA. However, its projection to *B* replaces γ transition with an ε-transition, and its projection to *C* replaces the α transition with an ε-transition. After the determinization, the automaton for peer *C* consists of one initial and one final state, and a single transition labeled γ 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 γ but the receiver, i.e., *A*, is not yet ready to receive the message. Hence, P_b in Figure 1 does not satisfy the synchronous compatibility condition. It can be verified that P_a and P_c in Figure 1 satisfy the synchronous compatibility condition.

(3) Autonomy condition: A conversation protocol satisfies the autonomy condition if at every state each peer is able to do exactly one of the following: to send a message, to receive a message, or to 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 Figure 1. The two transitions labeled α and β originating from the initial state violate the autonomy condition, because at the initial state, peer A can either send or receive. However, notice that the conversation protocols P_a and P_b in Figure 1 satisfy the autonomy condition.

We have the following results concerning the three conditions introduced above.

Theorem 1. If a conversation protocol (without message contents) satisfies the lossless join condition, synchronous compatibility condition, and autonomy condition, it is realized by its projections to each peer.

The key proof idea of Theorem 1 (Fu, Bultan, & Su, 2003) is that when a conversation protocol satisfies the realizability conditions, the composition of its projections to peers has a property called "eager consumption", which ensures that during any execution (interleaving) of the peers, whenever a peer sends out a message it is not in a final state and its input queue is always empty, i.e., each incoming message is consumed "eagerly" before any send action is taken by its receiver. When a composition satisfies the "eager consumption" property, for any conversation generated by the composition (using the asynchronous communication semantics), its projection to a peer is an accepted word by that peer FSA. This naturally leads to the conclusion that the conversation set is in the join of the languages accepted by all peers. If the conversation protocol is lossless join, the conversation set generated using the asynchronous semantics is the same as the one generated using the synchronous communication. Therefore a conversation protocol is realizable if the three sufficient conditions are satisfied.

THE GUARDED AUTOMATA MODEL

The abstract model of conversation protocol presented in the previous section is still not sufficient for real-world practice, because messages exchanged among web services are simply abstract "message classes" without contents. In this section we present a formal model, called the "guarded automata" (GA) model, which takes message contents into consideration. We begin this section by defining the composition schema (i.e., the structure of a web service composition). Then we present the formal models for both the top-down and bottom-up specification approaches, namely, conversation protocols and web service compositions.

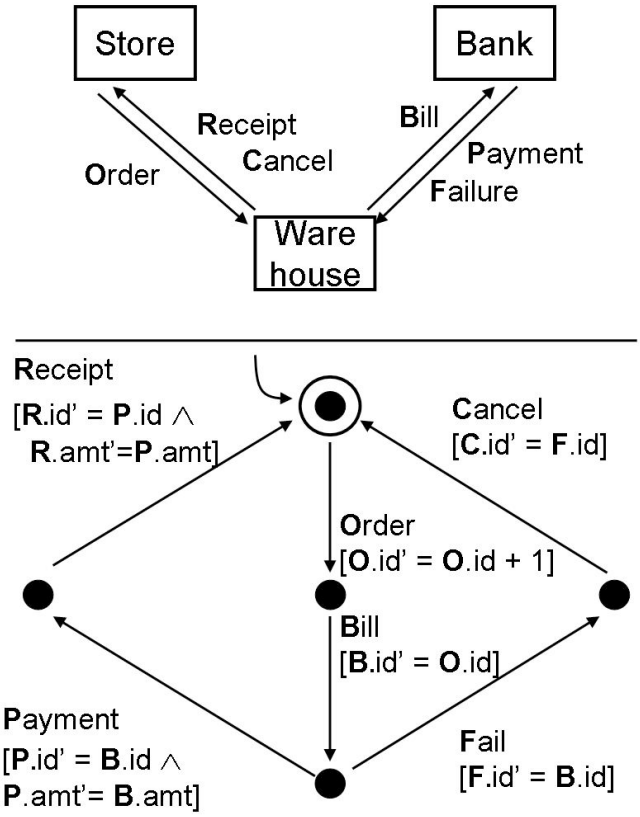

Figure 2. The composition schema and the conversation protocol for a simplified warehouse example.

Composition Schema

A composition schema entails the information about peers, message classes and the message domains for each message class. Formally, a composition schema is defined as follows:

Definition 1 (Composition Schema): A composition schema is a tuple (P, M, Σ) where P is the finite set of peers, M is the finite set of message classes, and Σ is the set of messages. Each message class $c \in M$ has a finite set of attributes and each attribute has a static data type. Each message $m \in \Sigma$ is an instance of a message class. Thus the message alphabet Σ can be defined as follows:

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

where DOM(c) is the domain of the message class c. Notice that Σ may be a finite or an infinite set, which is determined by the domains of the message classes.

Shown in Figure 2 is 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. Let us assume that each of **B**ill, **P**ayment, and **R**eceipt has two integer attributes id and amount (shortened by "amt"), and all other message classes in Figure 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, **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**. Note that, the domains of message classes **B**ill, **P**ayment, and **R**eceipt are Z × Z, where Z is the set of integers.

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.

Definition 2 (Conversation Protocol). A conversation protocol is a tuple $R = ((P, M, \Sigma), A)$, where (P, M, Σ) is a composition schema, and A is a **guarded automaton** (GA). The guarded automaton A is a tuple $(M, \Sigma, T, s, F, \delta)$ where M and Σ are the set of message classes and messages respectively, T is the finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is the set of final states, and δ is the transition relation. Each transition $\tau \in \delta$ is in the form $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of τ, $c \in M$ is a message class and g is the

guard of the transition. A guard g is a predicate on the attributes of the message that is being sent (which are denoted by the attribute names with apostrophe) and the attributes of the *latest* instances of the message classes (which are denoted by the attribute names without apostrophe) that are received or sent by the peer involved.

During a run of a guarded automaton, a transition is taken only if the guard evaluates to true. For example, in Figure 2, the guard of the transition that sends an **O**rder is:

$$\texttt{Order.id}' = \texttt{Order.id}+1.$$

The guard expresses that the `id` attribute of an **O**rder message is incremented by 1 after executing the transition. Notice that in the above formula "`id`′" refers to the value of the next **O**`rder` message being sent, and "`id`" refers to the value of the id attribute of the latest **O**`rder` message. With the use of primed forms of variables, the symbol "=" in guards stands for the "equality" instead of "assignment". The declarative semantics (instead of procedural semantics) used here is more convenient for symbolic analysis.

By studying the semantics of transition guards in Figure 2, it is not hard to see that the conversation protocol in Figure 2 describes the following desired message exchange sequence of the warehouse example: an **O**`rder` is sent by the Store to the Warehouse, and then the Warehouse sends a **B**`ill` to the Bank. The Bank either responds with a **P**`ayment` or rejects with a **F**`ail` message. Finally the Warehouse issues a **R**`eceipt` or a **C**`ancel` message. The guards determine the contents of the messages. For example, the `id` and the `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 $A = (M, \Sigma, T, s, F, \delta)$, a *run* of A is a path which starts from the initial state s and ends at a final state in F. A message sequence $w \in \Sigma^*$ is

accepted by *A* if there exists a corresponding run. For example, it is not hard to infer that the following is a message sequence accepted by the GA in Figure 2:

O(0), **B**(0, 100), **P**(0, 100),**R**(0, 100),**O**(1), **B**(1, 200), **F**(1),**C**(1).

Given a conversation protocol $R = ((P, M, \Sigma), A)$, its language is defined as $L(R) = L(A)$. In another word, the conversation set defined by a conversation protocol is the language accepted by its guarded automaton specification.

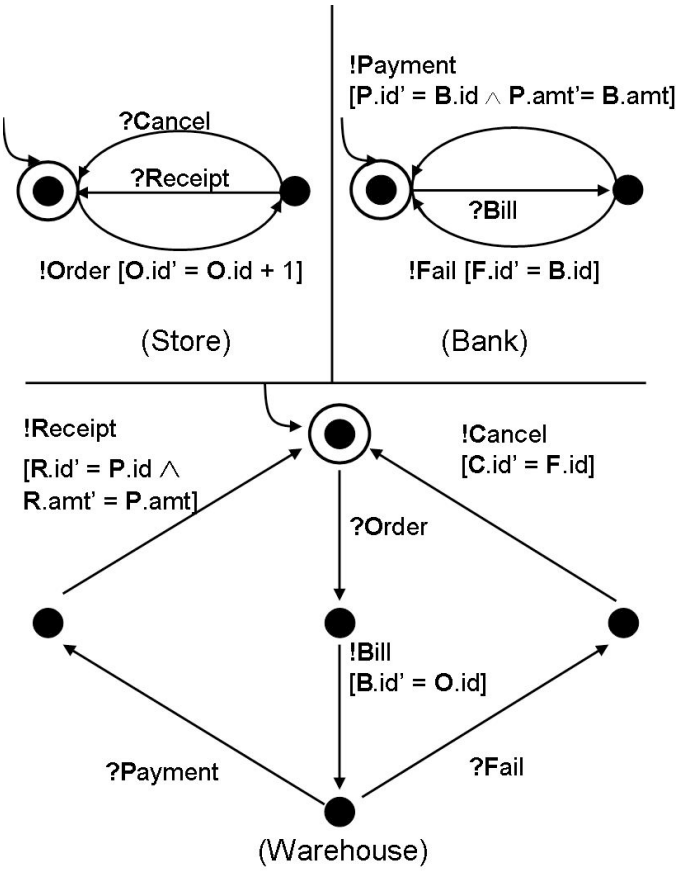

Figure 3. A realization of the conversation protocol in Figure 2.

Web Service Composition

Bottom-up web service compositions can be also specified using 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 as follows:

Definition 3 (Web Service Composition). A *web service composition* is a tuple $S = ((P, M, \Sigma), A_1, ..., A_n)$, where (P, M, Σ) is the composition schema, $n = |P|$, and for each $i \in [1..n]$, A_i is the peer implementation for $p_i \in P$. Each A_i is a tuple $(M_i, \Sigma_i, T_i, s_i, F_i, \delta_i)$ where M_i, Σ_i, T_i, s_i, F_i, and δ_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 ε-transition $(t, (\varepsilon, g_3), t')$, where $t, t' \in T_i$, $\alpha \in M_i^{\text{out}}$, $\beta \in M_i^{\text{in}}$, and g_1, g_2, and g_3 are predicates.

The send and receive transitions are denoted using symbols "!" and "?", respectively. In an ε-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 $(t, (?\beta, g_2), t')$, its guard determines whether the transition can take place based on the contents of the latest messages as well as the class and the contents of the message at the head of the queue. For a send transition $(t, (!\alpha, g_1), t')$, the guard g_1 determines not only the transition condition but also the contents of the message being sent. For example, Figure 3 shows a web service composition which realizes the conversation protocol in Figure 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. A global configuration contains the information about the local state and the queue contents of each peer, as well as the latest sent and received copies for each message class (so that the 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 is 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 ε-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. Let $C(S)$ denote the set of conversations of a web service composition S. We say S *realizes* a conversation protocol R if $C(S) = L(R)$.

CONVERSION BETWEEN TOP-DOWN AND BOTTOM-UP SPECIFICATIONS

One interesting question concerning the specification of web service compositions is if it is possible to convert a specification from top-down specification to bottom-up, and vice versa. This section introduces two operations to achieve this goal, namely "projection" and "product"

operations. The projection of a conversation protocol produces a web service composition, and the product of a web service composition produces a conversation protocol. However, the projection of a conversation protocol is not guaranteed to generate exactly the same set of conversations as specified by the protocol. Similarly, the 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 conditions (such as the ones presented in the skeleton analysis in later sections) are satisfied, can the projection and the product operations be used to convert from one specification approach to the other while preserving the semantics (i.e., the set of conversations).

Product

Intuitively, the product operation is to construct the synchronous composition of a set of guarded automata. In synchronous composition, each send operation and the corresponding receive operation have to be executed simultaneously. Formally, the product operation is defined as follows:

Definition 4 (Product). Let $S = ((P, M, \Sigma), A_1, ..., A_n)$ be a web service composition, where for each $i \in [1..n]$, A_i is a tuple $(M_i, \Sigma_i, T_i, s_i, F_i, \delta_i)$. The *product* of all peers in S is a GA $A' = (M, \Sigma, T', s', F', \delta')$, where each state $t' \in T'$ is a tuple $(t_1, ..., t_n)$, such that for each $i \in [1..n]$, t_i is a state of peer A_i. The initial state s' of A' corresponds to the tuple $(s_1, ..., s_n)$, and a final state in F' corresponds to a tuple $(f_1, ..., f_n)$ where for each $i \in [1..n]$, f_i is a final state of A_i. Let ϱ map each state $t' \in T'$ to the corresponding tuple, and further let $\varrho(t')[i]$denote the i'th element of $\varrho(t')$. For each pair of states t and t' in T', a transition $(t, (m, g'), t')$ is included in δ' 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)

- $\varrho(t)[i] = t_i$, and
- $\varrho(t')[i] = t'_i$, and
- $\varrho(t)[j] = t_j$, and
- $\varrho(t')[j] = t'_j$, and
- for each $k \neq i \wedge k \neq j$, $\varrho(t')[k] = \varrho(t)[k]$, and

2) (both guards need to hold) Let $g' = g_i \wedge g_j$, then g' must be satisfiable.

Clearly, by the above definition, the construction of the 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 from the source state simultaneously. Obviously, the construction can always terminate because the number of transitions and states of each peers is finite. However, the algorithm requires that none of the peer implementations should have ε-transitions. We will present the algorithm to remove ε-transitions later in this section.

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 to have an exact projection. For GA conversation protocols with infinite message alphabet, several "coarse" approximation algorithms exist. We start our discussion with the following classification of conversation protocols.

Definition 5 (FC and IC Protocol). A conversation protocol $R = ((P, M, \Sigma), A)$ is called an *Infinite Content (IC)* conversation protocol if Σ is an infinite set; otherwise R is called a *Finite*

Content (FC) conversation protocol. Similarly, a guarded automaton is either an IC-GA or FC-GA and a web service composition is either an IC composition or a FC composition.

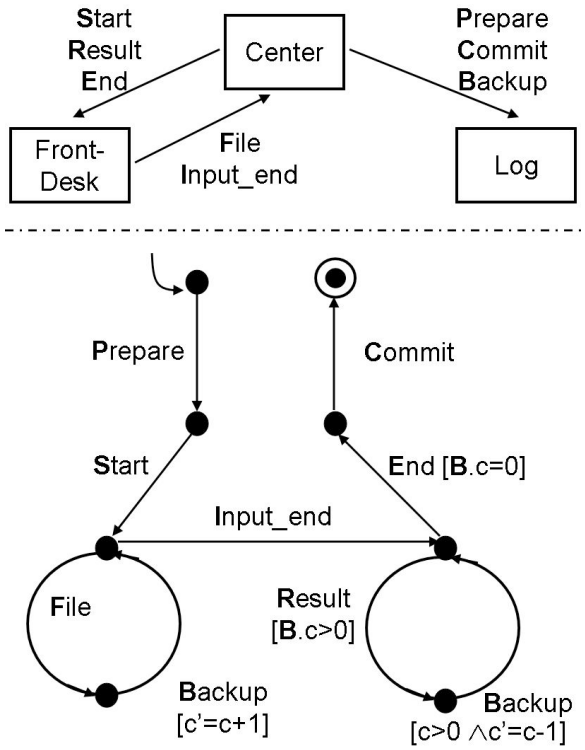

Figure 4. The Infinite Content (IC) Conversation Protocol which does not have an accurate projection to each peer.

An IC conversation protocol may not have an exact projection, as shown in the following example. Figure 4 presents an IC conversation protocol (call it *R*) which involves three peers Front-Desk, processing Center, and Log center. The processing Center processes files from Front-Desk in a batch sequential mode. For each incoming file, a backup of the request is recorded in the Log center and another backup is also created for the corresponding result.

The interaction between the three peers works as follows. The processing Center first sends a message **P**`repare` and **S**`tart` to the Front-Desk and Log center respectively to initiate

the process. Then the Front-Desk sends a collection of **F**iles one by one to the processing Center. For each incoming **F**ile, a corresponding **B**ackup copy is sent to Log center. The **B**ackup message has a counter named "c" to count the number of messages. When Front-Desk has sent out all the cases, it sends out an **I**nput_end message to mark the end of the data input process. The processing Center then processes the cases one by one, sends out a corresponding **R**esult message to Front-Desk, accompanied with a **B**ackup message for each **R**esult. The counter attribute of the **B**ackup message for each **R**esult is decremented by one each time until its values reaches zero. Finally, the processing Center sends out the **E**nd and **C**ommit message to inform both the Front-Desk and Log center to complete the interaction. Here all messages do not have contents except the **B**ackup message. If we use the first character to represent each message, obviously the conversation set can be represented using the following formula:

$$\mathbf{PS(FB)^{n}I(RB)^{n}EC}$$

If we project this conversation set to peer Log center, it is not hard to see that the projection is the following set $\{ \mathbf{PB^{n}B^{n}C} \mid n \geq 0\}$, which is obviously a context free language. However, since none of the messages (sent or received) by peer Log center has message contents, any GA with the alphabet of Log center is essentially a standard FSA, and it cannot accept the projection language, which is not regular.

On the contrary, for FC conversation protocols, it is always possible to construct a corresponding projected composition S_P^{PROJ} where each peer implementation of S_P^{PROJ} is an "exact" projection of P. Given a conversation protocol $R = ((P, M, \Sigma), A)$ where $n = |P|$. Since Σ is finite, we can easily construct a standard FSA A' from A such that $L(A') = L(A)$, where the alphabet of A' is the message alphabet Σ of A. The construction of A' is essentially an exploration

of all reachable global configurations of A. Note that, since Σ is finite, the number of global configurations of A is finite; hence, the size of A' is finite. Let the projection of A' to each peer be A_1, ..., A_n, respectively. Obviously, each of the standard FSA A_1, ..., A_n can be converted into an equivalent GA by associating dummy guards with each transition.

The construction of S_R^{PROJ}, however, can be very costly—it requires essentially a reachability analysis of the state space of the FC conversation protocol. In Figure 5, we present a light-weight projection algorithm, which is not "exact", but works for both FC and IC conversation protocols.

Procedure ProjectGA (((*P*, *M*, Σ), *A*), *i*): GA
Begin
  Let A' = (M, Σ, T, s_0, F, δ) be a copy of A.
  Substitute each (q_1, (a, g_1), q_2)
      where $a \notin M_i$ with (q_1, (ε, g_1'), q_2).
  Substitute each (q_1, (a, g_2), q_2)
      where $a \in M_i^{\text{in}}$ with (q_1, (?a, g_2'), q_2).
  Substitute each (q_1, (a, g_3), q_2)
      where $a \in M_i^{\text{out}}$ with (q_1, (!a, g_3'), q_2).
  // Generate g_1', g_2' and g_3' using **Coarse Processing 1** or **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, e.g., $g_1' \equiv \exists_{\{m \mid m \notin M_i\}} g_1$
    // g_3' = g_3.
  return A'.
End

Figure 5. Algorithm of coarse projection from a Guarded Automaton A to peer i.

Given a GA protocol and a peer to project to, the *coarse projection* algorithm in Figure 5 simply replaces each transition that is not related to the peer with ε-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 ε-transitions and the 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 the guards. The following example illustrates the existential quantification operation.

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 $m_1 \in M_3^{\text{out}} \cap M_1^{\text{in}}$, $m_2 \notin M_1$, and

$$g \sqsupset m_1.\text{id} + m_2.\text{id} < 3 \wedge m_2.\text{id} > 0.$$

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

Given a GA conversation protocol R, the web service composition generated using coarse-1 and coarse-2 processing algorithms are denoted as $S_R^{\text{PROJ,C1}}$ and $S_R^{\text{PROJ,C2}}$, respectively. Clearly $S_R^{\text{PROJ,C1}}$ and $S_R^{\text{PROJ,C2}}$ have the following relationship:

Given a conversation protocol $R = ((P, M, \Sigma), A)$, and its projections $S_R^{\text{PROJ,C1}}$ and $S_R^{\text{PROJ,C2}}$. For each $1 \le i \le |P|$, let A_i^{C1} and A_i^{C2} be the peer implementation of p_i in $S_R^{\text{PROJ,C1}}$ and $S_R^{\text{PROJ,C2}}$ (resp.). Then, the following holds: $\pi_i(L(R)) \subseteq L(A_i^{\text{C2}}) \subseteq L(A_i^{\text{C1}})$, i.e., the actual projection language is contained by the resulting set produced by the approximation algorithms.

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

Figure 6. Elimination of ε-transitions for Guarded Automata

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) to eliminate the ε-transitions, and (2) to determinize the result from step (1).

The ε-transition elimination algorithm is presented in Figure 6, which is an extension of the ε-transition elimination algorithm for standard FSA. It first collects a set of nodes which are reachable via ε-paths for each node. Then for each pair of nodes which are connected by an ε-path plus one normal transition, the algorithm packs the guards associated with the ε-path to the non ε-transition so that the ε-path can be eliminated. This operation is accomplished at line 11 of Figure 6. The transition $(t, (m, g'), t'')$ is a replacement for a set of paths, where each path is a concatenation of the transition $(t, (m, g), t')$ and an ε-path from t' to t''. The guard g' has to be the conjunction of g and g'' where g'' is the disjunction of the conjunctions of guards along each ε-path from t' to t''. Note that for each ε-transition, its guard is only a "transition condition" which

does not affect the message instance vector in a GA configuration, according to the guarded automata model. Hence it suffices to consider those non-redundant paths. Since the number of non-redundant paths is finite, the algorithm in Figure 6 will always terminate.

```
1. Procedure DeterminizeGA(A): GA
2. Begin
3.   Let A′ = (M, Σ, T, s, F, δ) be a copy of ElimGA(A).
4.   Mark all states in T as "unprocessed".
5.   For each "unprocessed" state t ∈ T Do
6.     For each message class m ∈ M Do
7.       Let {τ1, ..., τk} include each τi = (t, (m, gi), ti′)
             which starts from t and sends m.
8.       mark each τi as "toRemove"
9.       For each c = μ1 ∧ ... ∧ μk where μi is gi or ¬gi
           (however c≠¬g1 ∧ ¬g2 ... ∧ ¬gk ) 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 δ.
13.            For each j ∈ [1..k] s.t. gj appears in c Do
14.              For each transition τ′= (ti′, (m′, g′), tj″)
                 from ti′ and each τ″ = (s″, (m″, g″), ti′) to ti′ do
15.                  include (s′, (m′, g′), tj″) in δ
                     include (s″, (m″, g″),s′) in δ
16.                  mark τ′, τ″ as "toRemove".
                     mark ti′ as "toRemove".
17.              End For
18.            End For
19.          End If
20.      End For
21.    End For
22.  End For
23.  remove all states and transitions marked as "toRemove".
24.  return A′
25. End
```

Figure 7. Determinization of Guarded Automata

Figure 7 presents the determinization algorithm for a guarded automaton, which is rather different than the determinization algorithm for a standard FSA. The key idea is the part between lines 9 and 20, where for each state and each message class, we collect all the 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, and let their guards be g_1 and g_2 respectively. Three new transitions $g_1 \wedge g_2$, $g_1 \wedge \neg g_2$, and $\neg g_1 \wedge g_2$ will be generated, and the two original transitions are removed from the transition relation. It is not hard to see that for each word $w \in L(A')$, where 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, for example,

$$g_1 \wedge g_2 \vee g_1 \wedge \neg g_2 \vee \neg g_1 \wedge g_2 = g_1 \vee g_2,$$

each A is equivalent to its determinization A' (after applying DeterminizeGA), i.e., $L(A) = L(A')$.

SKELETON ANALYSIS

One interesting question concerning the GA model is: can the realizability analysis of a GA protocol be conducted using the techniques available on the standard FSA model? This section answers this question and shows that an abstract analysis method called **skeleton analysis** works for both FC and IC conversation protocols. We start with the definition of a skeleton.

Definition 6 (Skeleton). Given a GA $A = (M, \Sigma, T, s, F, \delta)$, its *skeleton*, denoted as *skeleton*(A), is a standard FSA (M, T, s, F, δ') where δ' is obtained from δ by replacing each transition $(s, (c, g), t)$ with (s, c, t).

Note that $L(skeleton(A)) \subseteq M^*$, while $L(A)$ is a subset of Σ^*. For a conversation protocol $R = ((P, M, \Sigma), A)$, we can always construct an FSA conversation protocol $((P, M), skeleton(A))$. We call this protocol the *skeleton protocol* of P. Now, one natural question is the following:

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

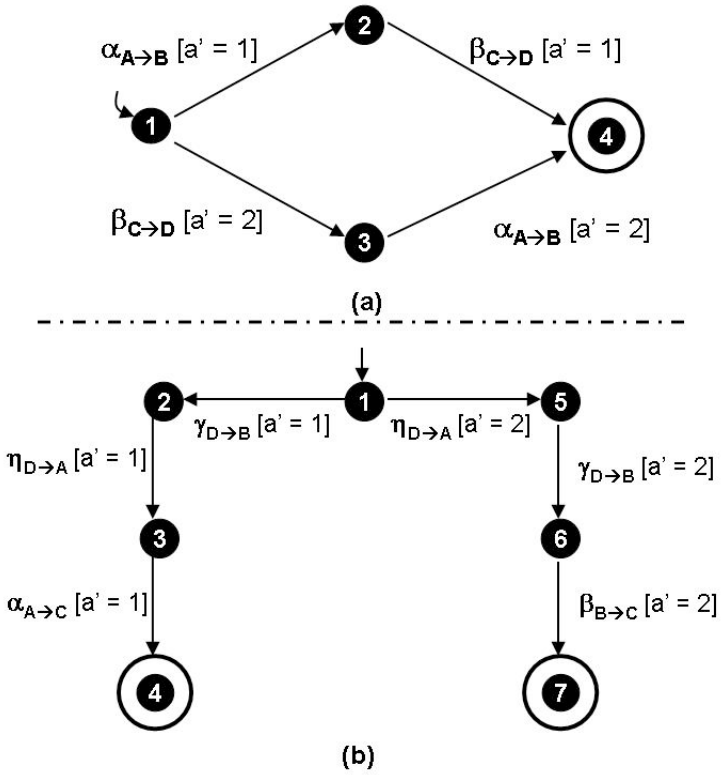

Figure 8. Two examples which demonstrate the relationship between a conversation protocol and its skeleton. (a) is an example where the GA conversation protocol is not realizable but its skeleton is. (b) is an example of realizable GA conversation protocol which has a non-realizable skeleton.

We can easily find counter examples against the above conjecture. As shown in Figure 8(a), four peers *A*, *B*, *C*, *D* are involved in a guarded conversation protocol. Message α *is* from *A* to *B* and β is from *C* to *D*. Both message classes have an attribute `a`. The protocol specifies two possible conversations $\alpha(1)\beta(1)$, and $\beta(2)\alpha(2)$, where the "1" and "2" are values of attribute `a` in

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

Interestingly, there exist examples where the skeleton is not realizable but the guarded conversation protocol is. In addition, the lossless join, and synchronous compatibility conditions of a guarded conversation protocol are not implied by the corresponding properties satisfied by its skeleton, and vice versa.

For example, the GA protocol in Figure 8(a) is not lossless join, however its skeleton is. The guarded protocol in Figure 8(b) is lossless join, however its skeleton is not. As shown in Figure 8(b), there are four peers *A, B, C, D,* 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 the attribute `a` (1 for left branch, 2 for right branch). Then *A* and *B* know who is going to send the last message (α or β), so there is no ambiguity. It can be verified that the protocol is lossless join. However the skeleton is obviously not lossless join, because $\eta\gamma\alpha$ is included in its join closure.

The counter examples in Figure 8 seem to suggest pessimistic results -- 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. We now introduce a fourth realizability condition to restrict a conversation protocol so that it can be realized by S_R^{PROJ}, $S_R^{\mathrm{PROJ,C1}}$, and $S_R^{\mathrm{PROJ,C2}}$ when its skeleton satisfies the three realizability conditions discussed above.

Definition 7 (Deterministic Guard Condition). Let $R = ((P, M, \Sigma), A)$ be a conversation protocol where $A = (M, \Sigma, T, s, F, \delta)$. R is said to satisfy the *deterministic guard 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 is 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 and w' where a partial run of w reaches t_1, and a partial run of w' reaches t_2, and $\pi_i(\pi_{\text{TYPE}}(w)) = \pi_i(\pi_{\text{TYPE}}(w'))$.

Here the operation π_{TYPE} projects a conversation to a sequence of message classes by replacing each message in the conversation with its message class. Intuitively, the deterministic guard 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 guard condition proceeds as follows: given a conversation protocol R, obtain its coarse-1 projection $S_R^{\text{PROJ,C1}}$, and let $S_R^{\text{PROJ,C1}} = ((P, M, \Sigma), A_1, ..., A_n)$. For each $i \in [1..n]$, regard A_i as a standard FSA, and get its equivalent deterministic FSA (let it be A_i'). Now each state t in A_i' corresponds to a set of states in A_i, and let it be represented by $T(t)$. We examine each state t in 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 a message with class c. We require that all guards collected for a state/message class pair (t, c) should be equivalent.

Running the algorithm on Figure 8(a) leads to the result that Figure 8(a) violates the deterministic guard condition, because (intuitively) peer A has two different guards when sending out α at the initial state. Formally, to show that the deterministic guard condition is

violated, we can find two transitions (t_1, (α, [$a' = 1$]), t_2) and (t_3, (α, [$a' = 2$]), t_4), and two words $w = \varepsilon$ and $w' = \beta(2)$ that lead to the states t_1 and t_3, respectively. Since a run of w reaches t_1, a run of w' *reaches* t_3, and $\pi_A(\pi_{\text{TYPE}}(w)) = \pi_A(\pi_{\text{TYPE}}(w')) = \varepsilon$. By Definition 7, the guards of the two transitions should be equivalent. However, they are not equivalent, which violates the deterministic guard condition.

Theorem 2. A conversation protocol R is realized by S_R^{PROJ}, $S_R^{\text{PROJ,C1}}$, and $S_R^{\text{PROJ,C2}}$ if it satisfies the deterministic guard condition, and its skeleton protocol satisfies the lossless join, synchronous compatibility and autonomy conditions.

The main proof idea of Theorem 2 is as follows. First, we argue that if the skeleton protocol satisfies the synchronous compatibility and autonomy conditions, then during any (complete or partial) run of $S_R^{\text{D,PROJ,C1}}$, each message is consumed "eagerly", i.e., when the input queue is not empty, a peer never sends out a message or terminates.

The "eager consumption" argument can be proved using proof by contradiction (Fu, Bultan, Su, 2003). Assume that there is a partial run against this argument, i.e., we can find a corresponding partial run of the skeleton composition of $S_R^{\text{D,PROJ,C1}}$ (which consists of the skeletons of each peer of $S_R^{\text{D,PROJ,C1}}$) 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 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 $C(S_R^{\mathrm{D,PROJ,C1}}) \subseteq L(R)$, as $L(R) \subseteq C(S_R^{\mathrm{D,PROJ,C1}})$ is obvious. Let $R = ((P, M, \Sigma), A)$ and let $S_R^{\mathrm{D,PROJ,C1}} = ((P, M, \Sigma), A_1, ..., A_n)$. Given a word $w \in C(S_R^{\mathrm{D,PROJ,C1}})$, and γ be the corresponding run, we can always construct a run γ' of A to recognize w. Since $\pi_i(w)$ is accepted by each peer A_i, $\pi_i(\pi_{\mathrm{TYPE}}(w))$ is accepted by *skeleton*(A_i). Because *skeleton*(A) is lossless join, it follows that $\pi_{\mathrm{TYPE}}(w)$ is accepted by *skeleton*(A), and let $T : \tau_1\tau_2...\tau_{|w|}$ be the path of *skeleton*(A) traversed to accept $\pi_{\mathrm{TYPE}}(w)$. Since each transition in *skeleton*(A) is the result of dropping the guard of a corresponding transition, we can have a corresponding path T' in 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 2, 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).

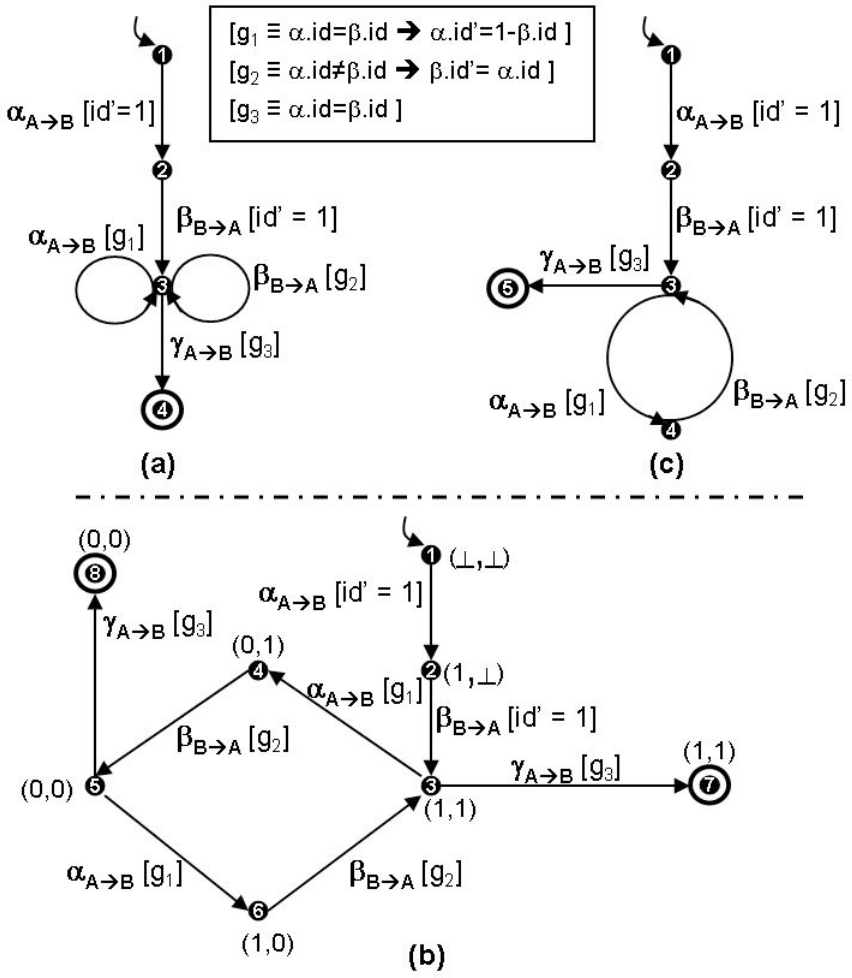

Figure 9. Alternating Bit Protocol

SYMBOLIC ANALYSIS

Sometimes skeleton analysis may be too coarse and fail to show the realizability of a realizable conversation protocol. For example, Figure 9(a) presents an alternating bit protocol which is realizable. However, the skeleton analysis presented in the previous section fails to show its realizability.

Let A_a, A_b, A_c be the three conversation protocols shown in Figure 9. The conversation protocol A_a consists of two peers *A* and *B*. Message class α is a request, and message class β is an acknowledgment. Both message classes contain an attribute called `id`. Message class γ is

used by *A* to notify *B* the end of conversation. The protocol states that the `id` attribute of α should alternate between 0 and 1, and every acknowledgment β must have the matching `id`. It is clear that the conversation protocol is non-ambiguous and realizable; however, the skeleton analysis fails to recognize it.

Clearly the projection of *skeleton*(A_a) to peer *A* does not satisfy the autonomy condition, because at state 3, there are both input and output transitions. However, A_a is actually autonomous. If we explore each configuration of A_a, we get A_b, the "equivalent" conversation protocol of A_a. The pair of values associated with each state in A_b stands for the `id` attribute of α and β. It is obvious that A_b satisfies the autonomy condition, and hence A_a should satisfy autonomy as well. In fact to prove that A_a is autonomous we do not even have to explore each of its configurations like A_b. As we will show later, it suffices to show A_c is autonomous.

Analysis of Autonomy Using Iterative Refinement

The examples in Figure 9 motivate the analysis of the autonomous condition using iterative refinement (Fu, Bultan, Su, 2004c, 2005a) as follows: Given a conversation protocol *A*, we can first check its skeleton. If the skeleton analysis fails, we can refine the protocol (e.g. refine A_a and get 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 A_b to 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, Δ) where *M* is the set of labels, *T* is the set of states, *s* the initial state, and Δ is 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, Δ) can be regarded as a transition system of (M, T, s, Δ); and a GA ($M, \Sigma, T, s, F, \Delta$) can be regarded as a transition system of the form (Σ, T', s', Δ') where T' contains all configurations of the GA, and Δ' defines the derivation relation between configurations.

Definition 8 (Simulation). A transition system $A' = (M', T', s', \Delta')$ is said to *simulate* another transition system $A = (M, T, s, \Delta)$, written as $A \leq A'$, if there exists a mapping $\varrho : T \square T'$ and $\xi : M \square M'$ such that for each (s, m, t) in Δ there is a $(\varrho(s), \xi(m), \varrho(t))$ in Δ'. Two transition systems A and A' are said to be *equivalent*, written as $A \cong A'$, if $A \leq A'$ and $A' \leq A$.

Intuitively a transition A' simulates A if we can find a corresponding action in A' for every action of A, i.e., A' can subsume the set of actions of A. For example, the following is true for the three conversation protocols A_a, A_b, A_c in Figure 9.

$$skeleton(A_b) \leq skeleton(A_c) \leq skeleton(A_a)$$

For example, in the simulation relation $skeleton(A_c) \leq skeleton(A_a)$, ϱ maps states 1, 2, 3, 4, 5 in $skeleton(A_c)$ to states 1, 2, 3, 3, 4 of $skeleton(A_a)$ respectively, and ξ is the identity function which maps each message class to itself. For another example, $A_a \leq skeleton(A_a)$, and $A_a \leq A_b \leq A_c$.

It is not hard to infer the following properties of simulation relation, detailed proof can be found in Fu, Bultan, and Su's work (2004c).

- For any GA A, $A \leq skeleton(A)$. In another word, the skeleton of a conversation protocol simulates the protocol.
- For each GA $A = (M, \Sigma, T, s, F, \Delta)$ on a finite alphabet Σ, there is a standard FSA on alphabet Σ such that $A \cong A'$. This can be easily achieved by exploring the configuration space of the GA protocol, which is finite.

- If $A \leq A'$ and A' is autonomous, then A is autonomous. The proof follows directly from the fact that each run of A has a corresponding run in A'. If during each run of A', autonomy condition is not violated, obviously any run of A will not violate it either.

From the above results, we can immediately infer the following:

Theorem 3. A GA conversation protocol is autonomous if its skeleton is autonomous.

```
Procedure AnalyzeAutonomy(A): List
Begin
   A′ = DeterminizeGA(A)
   While true do
      If skeleton of A′ is autonomous Then return null
      Find a pair (s, (m1, g1), t1), (s, (m2, g2), t2) violating autonomy
      (A′, trace) = Refine(A′, (s, (m1, g1), t1), (s, (m2, g2), t2))
      If trace≠ null Then return trace
   End While
End

Procedure Refine(A, (s, (m1, g1), t1), (s, (m2, g2), t2)) : (GA, List)
Begin
  If (Pre(g1) ∧ Pre(g2)) is satisfiable then
     Path = FindPath(A, s, Pre(g1) ∧ Pre(g2))
     If Path ≠ null then
        return (null, Path)
     End If
  End If
  Let A′ = (M′, T′, s0′, F′, Δ′) be a copy of A
  T′= T′ − {s} + {s1, s2}, F′ = F′ − {s} + {s1, s2} if s ∈ F′
  Substitute each (t, (mj, gj), s) in Δ′ s.t. mj ≠ m1 and mj ≠ m2
        with (t, (mj, gj), s1) and (t, (mj, gj), s2)
  Substitute each (s, (mj, gj), t) in Δ′ s.t. mj ≠ m1 and mj ≠ m2
        with (s1, (mj, gj), t) and (s2, (mj, gj), t)
  Substitute (s, (m1, g1), t1) in Δ′ with (s1, (m1, g1), t1)
  Substitute (s, (m2, g2), t2) in Δ′ with (s2, (m2, g2), t2)
  Remove all unreachable transitions
  return (A′, null)
End
```

Figure 10. The algorithm which examines the autonomy condition of a guarded conversation protocol using iterative refinement.

Based on Theorem 3 we have an error-trace guided symbolic analysis algorithm, presented in Figure 10. If the input GA is autonomous, procedure `AnalyzeAutonomy` returns null; otherwise it returns the error trace which is a list of configurations that eventually leads to the violation of the autonomy condition. `AnalyzeAutonomy` starts from the input GA, and refines it incrementally. During each cycle, it analyzes the skeleton of the current GA A'. If the skeleton is autonomous, by Theorem 3, the procedure simply returns and reports that the input GA is autonomous; otherwise, it identifies a pair of input/output transitions violating autonomy. For example, the two transitions starting at state 3 of Figure 9(a) will be identified. Then the `Refine` procedure is invoked to refine the current GA. This refinement process continues until the input GA is proved to be autonomous or a concrete error trace is found.

The bottom part of Figure 10 presents the algorithm of the `Refine` procedure. Its input includes two transitions (with guards g_1 and g_2 respectively) which lead to the violation of the autonomy condition on the skeleton. The `Refine` procedure 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., $\texttt{Pre}(g_1) \wedge \texttt{Pre}(g_2)$. If the conjunction is satisfiable, there is a possibility that at some configuration both transitions are enabled. Then we call the procedure `FindPath` to find a concrete error trace, which will be explained later. If the conjunction is not satisfiable, we can proceed to refine the GA. The basic idea is to split the source state of the two transitions into two states, each corresponds to the precondition of one guard in the input. The transitions are re-

wired correspondingly. Finally the procedure eliminates transitions that cannot be reached during any execution of the GA.

For example, if `Refine` is applied to Figure 9(a) and the two transitions starting at state 3, it first computes the conjunction of the 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, (states 3 and 4 in Fig. 9(c)), and transitions are modified accordingly. Finally, unreachable transitions are removed, which results in the GA in Figure 9(c).

Procedure FindPath(A, s, g_0): List
Begin
  Let $A = (M, T, s_0, F, \Delta)$
  Let T be
    $\bigcup_{(s_i,(m_j,g_k),s_i')\in\Delta} g_k \wedge state = s_i \wedge state' = s_i'$
  Stack path = new Stack()
  Let $g \equiv state = s \wedge g_0$
  g' = false
  stack.push(g)
  While $g \neq g'$ and $g \wedge state = s_0$ is not satisfiable **do**
    $g' = g$
    $g = (\exists M' (g_{M'/M} \wedge T)) \vee g'$
    path.push(g-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
        $(\exists M \text{ cvalue} \wedge T)_{M/M'} \wedge$ path[i+1]
    End For
    return ret
  End If
End

Fig. 11. Generation of a concrete error trace

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 ∧ b′ = 1`", its precondition is `Pre(`*g*`)` ◻∃`a′` ∃`b′ (a = 1∧ b′ = 1`), which is equivalent to "`a = 1`".

Figure 11 presents the algorithm to locate a concrete error trace. `FindPath` has three inputs: a GA *A*, a state *s* in *A*, and a symbolic constraint g_0. FindPath computes an error trace (a list of configurations) which starts from the initial state of *A*, and finally reaches *s* in a configuration satisfying constraint g_0.

The algorithm of `FindPath` is a variation of the standard symbolic backward reachability analysis used in model checking. It starts with the construction of a symbolic transition system *T* based on the control flow as well as the data semantics of *A*. Then given the target constraint g_0, the main loop computes the constraint which generates g_0 via transition system *T*. The loop terminates when it reaches the initial configuration, or it reaches a fixed point.

We use the following example to illustrate the symbolic backward analysis. In the example of Figure 9(a), if we redefine the guard g_2 as `◻id = ◻id ∧ ◻id′ = ◻id` , when procedure `Refine` is called on Figure 9(a), the conjunction of preconditions of g_1 and g_2, i.e., `◻id = ◻id` , is satisfiable. Then procedure `FindPath` is called with inputs Figure 9(a), state 3, and constraint `◻id = ◻id ∧ state = 3`. The while loop of `FindPath` eventually includes in variable `path` the following constraints:

1) `◻id = ◻id ∧ state = 3.`
2) `◻id = 1 ∧ state = 2.`
3) `state = 1.`

For example, given the formula (1) α.id = β.id ∧ state = 3 at state 3, the guard of the transition from state 2 to state 3 is α.id′ = α. id ∧ β.id′=1∧ state′ = 3 ∧ state=2, using the formula $(\exists M'\ (g_{M'/M} \wedge T))$ to compute its backward image at state 2 (where $g_{M'/M}$ means to substitute every message in M with its corresponding primed form) we have:

$(\exists M'\ (g_{M'/M} \wedge T))$
$\equiv (\exists M'\ ((\alpha.\text{id} = \beta.\text{id} \wedge \text{state} = 3)_{M'/M} \wedge (\alpha.\text{id}' = \alpha.\text{id} \wedge \beta.\text{id}' = 1 \wedge \text{state} = 2 \wedge \text{state}' = 3)))$
$\equiv (\exists M'\ (\alpha.\text{id}' = \beta.\text{id}' \wedge \text{state}' = 3 \wedge \alpha.\text{id}' = \alpha.\text{id} \wedge \beta.\text{id}' = 1 \wedge \text{state} = 2 \wedge \text{state}' = 3))$
$\equiv \alpha.\text{id} = 1 \wedge \text{state} = 2$

Then the order of `path` is reversed, and `cvalue` is randomly generated which satisfies constraint `state` = 1 and each message attribute has an exact value in `cvalue`. For example, let `cvalue` be □.id =1 ∧ □.id = 0 , then the list `ret` will record the following constraints:

1) □.id = 1 ∧ □.id = 0 ∧ state = 1.

2) □.id = 1 ∧ □.id = 0 ∧ state = 2.

3) □.id = 1 ∧ □.id = 1 ∧ state = 3.

It is not hard to see that the above list of system configurations captures an error trace leading to state 3 which violates the autonomy condition.

Complexity of the algorithms in Figures 10 and 11 depends on the data domains associated with the input GA. When the message alphabet is finite, they are guaranteed to terminate. For infinite domains, a constant loop limit can be used to terminate algorithms by force; however, the analysis is still conservative.

Symbolic Analysis of Other Realizability Conditions

It is interesting to ask: are there similar iterative analysis algorithms for the lossless join and synchronous compatibility conditions? The answer is negative, because the lossless join and

synchronous compatibility of a GA conversation protocol do not depend on those of its skeleton. In another word, there exists a GA conversation protocol which is lossless join and whose skeleton is not. There also exists a GA conversation protocol which is not lossless join however its skeleton is. Similar observation holds for synchronous compatibility.

In the following we introduce "conservative" symbolic analyses for these two conditions. We introduce the analysis for synchronous compatibility first. Recall the algorithm to check synchronous compatibility of a FSA conversation protocol. The protocol is projected to each peer and determinized (including ε-transition elimination). Then the Cartesian product is constructed from the deterministic projection to peers. Each state in the Cartesian product is examined. A state is called an illegal state if at the state some peer is not ready to receive a message that another peer is ready to send. Note that, the determinization of each peer projection is a necessary step. The analysis of synchronous compatibility for a GA conversation protocol follows a same procedure. However, we have to discuss two different cases on GA conversation protocols with finite or infinite domains. Given a FC conversation protocol R, we can always construct its exact equivalent FSA conversation protocol (let it be R'), and use the synchronous compatibility analysis for standard FSA protocols to analyze R'. 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 R, we can project it to each peer using coarse projection (either Coarse Processing 1 or Coarse Processing 2 in Figure 5). Then we determinize each peer in $S_R^{\mathrm{PROJ,C1}}$ (or $S_R^{\mathrm{PROJ,C2}}$) using the `DeterminizeGA` in Figure 7. We construct the product of those determinized GA. If no illegal state is found, the IC conversation protocol R is

synchronous compatible. The method is conservative, i.e., if an illegal state is found, R might still be synchronous compatible, because a coarse projection accepts a superset of the language accepted by the exact projection. However, if a conversation protocol is identified as synchronous compatible by the approximation algorithm, it is guaranteed to be truly synchronous compatible.

The analysis of lossless join condition is similar. Recall that each GA A can be regarded as a transition system, and can be represented symbolically. Let $T(A)$ denote the symbolic transition system derived from A. From the initial configuration of A, we can compute all the reachable configurations of $T(A)$, and let the set of reachable configurations be S^A. Given A_1 and A_2, the following statement is true:

$$(S^{A1} \wedge T(A_1) \Rightarrow S^{A2} \wedge T(A_2)) \Rightarrow (L(A_1) \subseteq L(A_2)).$$

Intuitively, the equation means that if A_2 as a transition system is a superset of A_1, i.e., for any reachable configuration, there are more enabled transitions in $T(A_2)$ than $T(A_1)$, then $L(A_2)$ should be a superset of $L(A_1)$. The equation naturally implies a symbolic analysis algorithm. Given a conversation protocol R (with finite or infinite domains), let its GA specification be A. We can project A using coarse projection. Then construct the product of $S_R^{\text{PROJ,C1}}$ (or $S_R^{\text{PROJ,C2}}$), and let it be A'. Then we construct $T(A)$ and $T(A')$, and compute S^A and $S^{A'}$. It is not hard to see that if $(S^{A'} \wedge T(A')) \Rightarrow (S^A \wedge T(A))$, we can conclude that R is lossless join. The above symbolic analysis algorithm is decidable when the domain is finite. When R has an infinite domain, we can simply use the approximate closure of S^A and $S^{A'}$, and it is still a conservative algorithm.

CONCLUSION

This chapter presents Bultan, Fu, Hull, and Su's discovery on the realizability problem of conversation protocols. The analysis can be conducted on two levels: the abstract level without data semantics and the concrete level with message contents. The chapter reveals the relationship between the realizability analyses on the two models. It is shown that 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. When skeleton analysis is not precise enough, refined symbolic realizability analyses can be used to improve both the accuracy and efficiency of the analysis.

The skeleton realizability analysis presented in this chapter has been implemented as a part of the Web Service Analysis Tool (**WSAT**) (Fu, Bultan, & Su, 2004d). The front-end of WSAT accepts industry web service standards such as WSDL and BPEL. The core analysis engine of WSAT is based on the intermediate representation GA. The back-end employs model checker **SPIN** (Holzmann, 1997) for verification. At the front-end, a translation algorithm from BPEL to GA is implemented. Then at the core analysis part, realizability analysis and another similar analysis called "synchronizability analysis" are implemented to avoid the difficulty of verification in the presence of 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, LTL verification at the back-end can be performed using the synchronous communication semantics instead of asynchronous

communication semantics. WSAT is applied to verify a wide range of examples, including conversation protocols converted from IBM Conversation Support Project (IBM, n.d.), five BPELS services from BPEL4WS standard and Collaxa.com, and the SAS example (Fu, Bultan, Su, 2004d). The empirical experiences suggest that the realizability conditions presented in this chapter can capture a large class of real-world web service designs.

REFERENCES

Abadi, M., Lamport, L., & Wolper, P. (1989). Realizable and unrealizable specifications of reactive systems. In *Proceedings of 16th International Colloquium on Automata, Languages and Programming*, 1–17.

Alur, A., Etessami, K., & Yannakakis, M. (2001). Realizability and verification of MSC graphs. In *Proceedings of 28th International Colloquium on Automata, Languages, and Programming*, 797–808.

Alur, R., McMillan, K. , & Peled, D. (2000). Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160, 167–188.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., et al, (2003), Business Process Execution Language for Web Services (BPEL) 1.1. Retrived from http://www.ibm.com/developerworks/library/wsbpel.

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

Betin-Can, A., Bultan, T., Fu, X. (2005). Design for verification for asynchronously communicating Web services, in *Proceedings of 14th international conference on World Wide Web (WWW).*

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

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

Canfora, G., & Di Penta, M. (2006). Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2), 10–17.

Clarke, E. M., Grumberg, O., & Peled, D. A. (2000). *Model Checking*. MIT Press.

Foster, H. , Uchitel, S., Magee, J. , & Kramer, J. (2003) Model-based verification of web service compositions. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering Conference (ASE)*, 152–161.

Fu, X., Bultan, T. & Su, J. (2003). Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proeedings of 8th International Conference on Implementation and Application of Automata (CIAA)*, 188–200.

Fu, X., Bultan, T., & Su, J. (2004a). Analysis of interacting web services. In *Proceedings of 13th International World Wide Web Conference (WWW)*, 621–630.

Fu, X., Bultan, T., & Su, J. (2004b). Model checking XML manipulating software. In *Proceedings of 2004 International Symosium on Software Testing and Analysis (ISSTA)*, 252–262.

Fu, X., Bultan, T., & Su, J. (2004c).Realizability of conversation protocols with message contents. In *Proceedings of 2004 IEEE International Conference on Web Services (ICWS)*, 96–103.

Fu, X., Bultan, T., & Su, J. (2004d). WSAT: A tool for formal analysis of web service compositions. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV)*, 510–514.

Fu, X., Bultan, T., & Su, J. (2004e). Conversation protocols: A formalism for specification and analysis of reactive electronic services. *Theoretical Computer Science*, 328(1-2), 19–37.

Fu, X., Bultan, T., & Su, J. (2005a). Realizability of conversation protocols with message contents (Extended version of the ICWS'04 paper), *International Journal of Web Services Research (JWSR)* 2(4), 68–93.

Fu, X., Bultan, T., & Su, J. (2005b). Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31 (12), 1042–1055.

Hanson, J. E., Nandi, P., & Levine, D. W. (2002). Conversation-enabled web services for agents and e-business. In *Proceedings of 2002 International Conference on Internet Computing (IC)*, 791–796.

Hanson, J. E., Nandi, P., & Kumaran, S. (2002). Conversation support for business process integration. In *Proceedings of 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 65–74.

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

IBM. (n.d.). Conversation Support Project. Retrieved Feb 14, 2007, from http://www.research.ibm.com/convsupport/.

ITU-T. (1994). Message Sequence Chart (MSC).*Geneva Recommendation Z.120.*

Microsoft. (n.d.). MicroSoft Message Queuing Service. Retrieved Feb 14, 2007, from http://www.microsoft.com/msmq/.

Narayanan, S. & Mcllraith, S. A. (2002) Simulation, verification and automated composition of web services. In *Proeedings of 11th International World Wide Web Conference (WWW)*, 77–88.

Pnueli, A., & Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, 179–190.

Sun. (n.d.) Java Message Service. Retrieved Feb 14, 2007, from http://java.sun.com/products/jms/.

World Wide Web Consortium. (2001). Web Services Description Language (WSDL) 1.1. Retrieved Feb 14, 2007, from http://www.w3.org/TR/wsdl, March 2001.

World Wide Web Consortium. (2002). Web Service Choreography Interface (WSCI) 1.0. Retrieved Feb 14, 2007, from http://www.w3.org/TR/2002/NOTE-wsci-20020808.