Realizability Analysis for Message-based Interactions Using Shared-State Projections

Sylvain Hallé
Université du Québec à Chicoutimi, Canada
shalle@acm.org

Tevfik Bultan[*]
University of California, Santa Barbara, USA
bultan@cs.ucsb.edu

ABSTRACT

The global interaction behavior in message-based systems can be specified as a finite-state machine defining acceptable sequences of messages exchanged by a group of peers. Realizability analysis determines if there exist local implementations for each peer, such that their composition produces exactly the intended global behavior. Although there are existing sufficient conditions for realizability, we show that these earlier results all fail for a particular class of specifications called arbitrary-initiator protocols. We present a novel algorithm for deciding realizability by computing a finite-state model that keeps track of the information about the global state of a conversation protocol that each peer can deduce from the messages it sends and receives. By searching for disagreements between each peer's deduced states, we provide a sound analysis for realizability that correctly classifies realizability of arbitrary-initiator protocols.

Categories and Subject Descriptors

B.4.3 [**I/O and Data Communications**]: Interconnections (Subsystems)—*Asynchronous/synchronous operation, Web technologies*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Tools, validation*

General Terms

Verification

Keywords

realizability, asynchronous communication, choreography

1. INTRODUCTION

Nowadays, many software systems consist of multiple components that execute concurrently, possibly on different machines. New trends in computing, such as service-oriented architecture (SOA), cloud computing, multi-core hardware, all point to more concurrency and distribution among the components of software systems in the future. In order to complete a task, components of a software system have to coordinate their executions by interacting with each other. A fundamental question is, what should be the interaction mechanism given the trend for increased level of concurrency and distribution? One emerging paradigm is message-based communication [3, 5, 8, 15, 18, 19, 21], where components interact with each other by sending and receiving messages.

This is particularly the case for SOA, which, through the use of SOAP and XML, enable interaction among existing services to form composite, value-added applications. A key factor in SOA is the loose coupling between the various parties involved in an interaction. Apart from a mutually-agreed protocol of interaction (called a conversation protocol [10,11] or a choreography specification [22] in SOA), each peer taking part in such a global "conversation" is an independent and self-contained unit.

To further loosen this coupling between potential partners, many web services are implemented using *asynchronous* communication. While in synchronous communication, the sender and receiver of a message are required to block their operation for the duration of the message transfer, in asynchronous communication this restriction is relaxed by treating sending and receiving as two separate events. The recipient of a message is no longer required to synchronize with the sender, and can rather buffer messages in an input queue and process them at an arbitrary later time.

Interestingly, the same type of message-based interactions appear in other domains as well. For example, Singularity is a new, experimental, operating system developed by Microsoft Research to explore new approaches to OS design [14]. One of its main goals is to improve the dependability of software systems by rethinking some design decisions that have largely governed operating system architecture to date. Process isolation is a chief design principle of the Singularity operating system. To achieve process isolation, certain constraints are enforced to ensure process independence. Among these is the rule that processes cannot share memory with each other or the kernel. All interprocess communication in Singularity, therefore, occurs via message passing over bidirectional conduits, called channels. Channels have two end points referred to as the client and the server that communicate with each other by sending and receiving messages. Communication through Singularity channels corresponds to asynchronous communication

[*]This work is supported by NSF grants CCF-0916112 and CCF-0716095, and by the Fonds québécois de recherche sur la nature et les technologies (FQRNT).

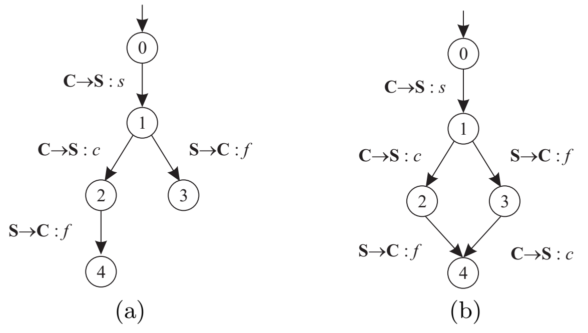


Figure 1: Two simple file transfer protocols.

via FIFO message queues. Moreover, in Singularity, each channel is governed by a channel contract [8]. A channel contract is a state machine that specifies the allowable ordering of messages between the client and the server. Hence, channel contracts are a type of conversation protocols and serve the same purpose that choreography specifications serve in SOA.

Asynchronous communication is appealing for simplifying the management of message exchanges in SOA or when isolating processes from each other in an operating system, but many natural assumptions about synchronous protocols no longer hold when message queuing is involved. For example, Figure 1a shows a specification for the control channel of a simple file transfer protocol between a client (C) and a server (S). After asking for the start of a transfer (s), the sequence of possible messages depends on who initiates the remaining part of the conversation. If C sends a "cancel transfer" message (c) first, then S should *reply* to C's request with a "transfer finished" message (f); on the other hand, no cancellation request should be issued by C once S notifies that the transfer is complete.

Albeit simple, this protocol presents problems. In the situation where S sends f first, but its message remains in transit for some time, C might believe the transfer is not finished and send c, thus violating the protocol and perhaps throwing S into an undefined state. This happens, even though each peer locally complies with the protocol. The "fault", in this case, is not any of the peers'; it is rather rooted in the fact that asynchronous communication is used. Protocols that exhibit this type of behavior are called *unrealizable* protocols.

The question of deciding whether a given protocol is realizable or not is an open problem. The presence of unbounded input queues makes the exhaustive search of all possible communication traces impossible. There has been two separate branches of research to address this problem and sufficient realizability conditions have been developed independently 1) through the analysis of the state machines characterizing the allowable conversations (e.g., [10, 11, 16]), or 2) by reformulating the problem within a type system that prevents the declaration of unrealizable protocols (e.g., [12,13]). These conditions are sound and can be used to formally prove the realizability, however, they are not complete, and can produce false positives. In particular, a "fixed" version of the file transfer protocol, shown in Figure 1b, will be identified as unrealizable by both approaches, although it actually *is* realizable. We show that this will be the case of any protocol containing a state with an *arbitrary initiator*, i.e., a state where more than one peer could send the next message and the protocol works fine for either case.

In this paper, we present a novel way of deciding the realizability of a protocol in the presence of asynchronous communication. Our technique is based on the observation that in a realizable protocol, a global observer can follow the conversation between the peers and, at any moment, determine precisely the global state of the protocol. Each individual peer, based on the messages it sends and receives, can only guess plausible values for that global state. We show how to compute an annotated version of the original protocol, specific to each peer, called *shared-state projections*, including such guesses. We demonstrate that, at any moment in a realizable protocol, there exists at least one state which all peers agree is a "believable" global state. Conversely, this entails that in many cases, unrealizable protocols have two of their peers straying until they have irreconcilable beliefs of the current global state. Since the number of protocol states is finite, the number of belief states is also finite, and searching for such "discordant" belief states can be done exhaustively.

The contributions of this paper are: 1) We provide a natural explanation of realizability based on the consistency of the observations of the different peers. 2) We give a new, sufficient condition to ensure realizability of a conversation protocol that can be evaluated automatically using shared-state projections. 3) We show that our realizability condition can correctly identify realizable arbitrary-initiator protocols, as opposed to earlier work based on conversations [10, 11] and session types [13] which always produce false positives for this class of protocols.

In Section 2, we review the concepts about conversation protocols and define realizability. In Section 3, we develop the concept of *shared-state channel system* and show how to compute shared-state projections. We also demonstrate the soundness of our method. In Section 4, we present a tool we developed to automatically compute belief projections and analyze its results on more than 100 real-world protocols. We show in particular how protocols deemed unrealizable with earlier approaches can be shown realizable with our new approach. In Section 5, we review the earlier results on realizability and show that they fail to correctly analyze arbitrary-initiator protocols. Finally, in Section 6 we conclude the paper and indicate further research directions.

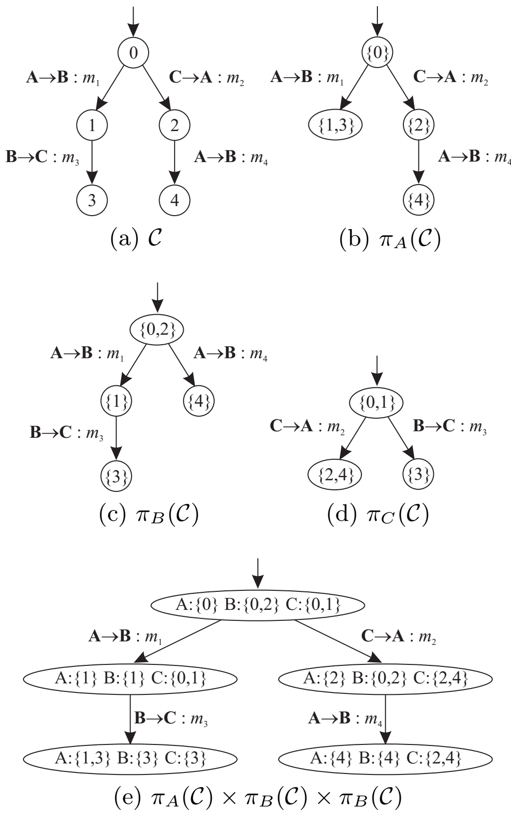
2. MODELING MESSAGE-BASED INTERACTIONS

In this section we first present a formal model for conversation protocols, which are finite state machines that are used to specify the set of allowable message sequences. Next we discuss channel systems, which is a formal model for a set of components (i.e. peers) that interact with asynchronous messaging. Finally, we formalize the realizability problem based on conversation protocols and channel systems.

2.1 Conversation Protocols

A natural way of specifying the allowable message-based interactions in a concurrent or distributed system is to use a finite state machine that serves as the agreed-upon protocol among the peers that interact with each other. Such specifications can be formalized as *conversation protocols* [10]:

Definition 1. *A* conversation protocol *is a quintuplet*



(a) $\mathcal{C}$ (b) $\pi_A(\mathcal{C})$

(c) $\pi_B(\mathcal{C})$ (d) $\pi_C(\mathcal{C})$

(e) $\pi_A(\mathcal{C}) \times \pi_B(\mathcal{C}) \times \pi_B(\mathcal{C})$

Figure 2: A conversation protocol $\mathcal{C}$ (a), its projection for peers A, B and C (b–d), and the product of these three projections (e).

$\mathcal{C} = (P, S, s_0, L, \delta)$ *where* $P = \{p_0, \ldots, \}$ *is a finite set of* peers, $S = \{s_0, \ldots, \}$ *is a finite set of* states, $s_0 \in S$ *is the initial state,* $L = \{\ell_0, \ldots\}$ *is finite a set of* message labels, *and* $\delta : S \times P \times L \times P \to S$ *is a transition function.*

A triplet (p, ℓ, p') is called a *message* and represents peer p sending the message with label ℓ to peer p'. The transition function is such that $\delta(s_m, p, \ell, p')$ returns the state s_n accessible from state s_m for that message. We extend the definition of δ over sets of states, such that for a subset $S' \subseteq S$, we have $\delta(S', p, \ell, p') = \bigcup_{s \in S'} \delta(s, p, \ell, p')$.

A sequence of messages is accepted by $\mathcal{C}$ as long as it corresponds to some valid path that starts from its initial state. We do not define explicit "accepting states" for conversation protocols (or equivalently we assume that all states are accepting). The *language* accepted by $\mathcal{C}$, noted $\mathcal{L}(\mathcal{C})$, is the set of message sequences that are accepted by $\mathcal{C}$.

A conversation protocol $\mathcal{C}$ can be *projected* to one of its peers p, noted as $\pi_p(\mathcal{C})$, which is obtained by replacing every transition where p is neither the sender nor the receiver by an ϵ-transition. For example, Figure 2b shows the projection of Figure 2a onto peer A. Since, in that protocol, the transition from state 1 to state 3 does not involve A, it becomes an "invisible" transition in the projection. One might then collapse any states linked by an ϵ-transition as a single, compound state whose label becomes a subset of states from the original conversation. For peer A, the ϵ-transition causes the collapse of states 1 and 3 into a single state, labeled $\{1, 3\}$.

Individual conversation protocols $\mathcal{C}_0, \ldots, \mathcal{C}_n$ can also be combined in a form of Cartesian product, noted $\mathcal{C}_0 \times \cdots \times \mathcal{C}_n$, similar to DFA product operation. Figure 2 shows examples of these concepts [4].

2.2 Channel Systems

A *channel system* is a formal model for a set of peers that interact with asynchronous messages. When a message is sent it is not immediately received as in synchronous communication, but, rather, it is stored in the input queue of the receiver and can be consumed by the receiver at an arbitrary later time.

One way of producing a channel system is by composing a set of conversation protocols $\mathcal{C}_0, \ldots, \mathcal{C}_n$ corresponding to peer behaviors. Intuitively, each peer possesses its own input message queue and its own finite-state control; it sends and receives messages according to its individual control state and queue contents. Channel systems formalize this type of behavior [2].

Let P be a set of peers, L be a set of message labels, and $\mathcal{C}_i = (P, S^i, s_0^i, L, \delta^i)$ $(0 \leq i \leq |P|)$ be a set of conversation protocols, one for each peer in P. For a set of messages $M = (P \times L \times P)$, we let M^* denote the set of finite words over this alphabet of triplets; these words represent the contents of each peer's input message queue. A *global state* $\dot{s}$ of $\dot{\mathcal{C}}$ is a pair $\langle(\dot{s}^0, \ldots, \dot{s}^{|P|}), w\rangle$, where $\dot{s_i} \in S^i$ and $w : P \to M^*$ associates each conversation protocol with a state and a message vector (denoting the contents of each peer's input message queue). A channel system is defined over these elements as follows:

Definition 2. *A* channel system $\dot{\mathcal{C}}$ *is a quintuplet* $(P, \dot{S}, \dot{s_0}, M, \dot{\delta})$ *where* $\dot{S} \subseteq (S^0 \times \cdots \times S^{|P|}) \times M^{*|P|}$ *is a finite set of control states,* $\dot{s_0} \in \dot{S}$ *is the initial control state,* $M = P \times L \times P$ *is a finite set of messages, and* $\dot{\delta} \subseteq \dot{S} \times (M \times \{!, ?\}) \times \dot{S}$ *is a finite set of transitions.*

The initial global state $\dot{s_0}$ of $\dot{\mathcal{C}}$ is the pair $\langle(s_0^0, \ldots, s_0^{|P|}), w_\epsilon\rangle$, where $w_\epsilon(p) = \epsilon$ for each peer p (i.e., each peer starts with an empty input message queue).

Since send and receive events are now decoupled from each other, the transition relation is defined over *directed messages* $m^\rightarrow$, which are tuples $(m, \star) \in M \times \{!, ?\}$, where m is a message and $\star$ is either "!" (representing the sending of m) or "?" (representing m being read from the recipient's input queue).

The relation $\dot{\delta}$ is defined as the smallest set of triplets $(\dot{s}, m^\rightarrow, \dot{s}')$ where $\dot{s} = (\dot{s_0}, \ldots, \dot{s_{|P|}})$ and $\dot{s}' = (\dot{s_0}', \ldots, \dot{s_{|P|}}')$ are control states, and $m^\rightarrow = ((p_m, \ell, p_n), \star)$ is a directed message, such that:

1. If $\langle \dot{s}, ((p_m, \ell, p_n), !), \dot{s}' \rangle \in \delta$, then the control states change from $\dot{s}$ to $\dot{s}'$ and (p_m, ℓ, p_n) is appended to the tail of channel $w(p_n)$; all other channels do not change. Moreover, $\dot{s_m}' = \delta^m(\dot{s_m})$ and all others $\dot{s_i}'$ do not change.

2. Similarly, if $\langle \dot{s}, ((p_m, \ell, p_n), ?), \dot{s}' \rangle \in \delta$, then the control states change from $\dot{s}$ to $\dot{s}'$ and (p_m, ℓ, p_n) is removed from the head of channel $w(p_n)$; all other channels do not change. Moreover, $\dot{s_n}' = \delta^n(\dot{s_n})$ and all others $\dot{s_i}'$ do not change.

As an example, Figure 3 shows a portion of the channel system obtained by composing the projections in Figure 2. Each state of this channel system lists the projection state for peers A, B, and C, followed by the contents of the queue for each peer, in the form $(p, w(p))$, where p is the peer's name and $w(p)$ is the contents of its queue.

From a trace of states $\dot{s}_0, \dots, \dot{s}_k$ in a channel system $\dot{\mathcal{C}}$, one can extract the corresponding sequence of directed messages $(m_0, \star_0), \dots, (m_{k-1}, \star_{k-1})$. The *send trace* is the subsequence obtained by keeping the directed messages m_i such that $\star_i =!$ and removing all the other directed messages. The set of all send traces produced by $\dot{\mathcal{C}}$ is denoted as $\mathcal{L}(\dot{\mathcal{C}})$; we call it the *language accepted by* $\dot{\mathcal{C}}$.

2.3 Realizability

There are six possible executions in the portion of channel system in Figure 3; these executions create five distinct send traces. Let us consider the trace shown in bold, which corresponds to the following sequence of events. First, peer C, being in state $\{0, 1\}$ of its projection, can move to its state $\{2, 4\}$ and send message m_2 to A's input queue, which corresponds to the transition to the subsequent state. Next, A, still being in state $\{0\}$ of its projection, chooses to place message m_1 in B's input queue, and moves to state $\{1, 3\}$. Finally, B consumes message m_1, and immediately sends m_3, ending in its projection state $\{3\}$.

This execution has several issues. First, in the last state of the channel system, A is in state $\{1, 3\}$ and does not expect any more messages, yet m_2 is waiting in its queue. Second, the send trace corresponding to this execution is not part of the conversation protocol from which the channel system was derived. As a matter of fact, from the six possible send traces generated by this portion of channel system, four of them are "spurious" message sequences not specified by the original conversation protocol.

THEOREM 1. *Let $\mathcal{C}$ be a conversation protocol and $\dot{\mathcal{C}}$ be the channel system obtained from $\pi_{p_0}(\mathcal{C}), \dots, \pi_{p_{|P|}}(\mathcal{C})$. Then $\mathcal{L}(\mathcal{C}) \subseteq \mathcal{L}(\dot{\mathcal{C}})$.*

PROOF. Let $\overline{s} = s_0, s_1, \dots$ be a state trace of $\mathcal{C}$. We build a trace $\dot{\overline{s}} = \dot{s_0}, \dot{s_1}, \dots$ from $\overline{s}$ as follows. We take $\dot{s_0}$ as the start state in Definition 2. For each $i \geq 0$, if $\langle s_i, (p_j, \ell, p_k), s_{i+1}\rangle \in \delta$ then:

- $\dot{s}_{2i+1} = \langle (S^{0\prime}, \dots, S^{|P|\prime}), w'\rangle$ is the global state obtained from $\dot{s}_{2i}$ by taking the transition labeled with the directed message $((p_j, \ell, p_k), !)$
- $\dot{s}_{2i+2} = \langle S^{0\prime\prime}, \dots, S^{|P|\prime\prime}, w''\rangle$, is the global state obtained from $\dot{s}_{2i+1}$ by taking the transition labeled with the directed message $((p_j, \ell, p_k), ?)$

Trivially, $\dot{\overline{s}}$ has the same send trace as $\overline{s}$; moreover its construction follows Definition 2 and is therefore a trace of global states in $\dot{\mathcal{C}}$. $\square$

The fact that the relation between $\mathcal{C}$ and $\dot{\mathcal{C}}$ is an inclusion, and not an equality, is the source of the problem described in the file transfer protocol of Figure 1a: the channel system obtained from the projections of that protocol onto peers C and S produces one send trace (s, f, c) that is not in the original specification. A protocol is said to be *realizable* when we have a strict equality:

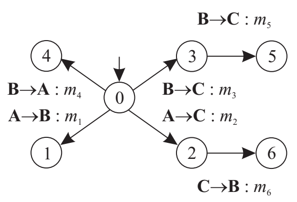


Figure 4: Deducing B's possible state from A's point of view in a conversation protocol.

DEFINITION 3 (REALIZABILITY). *A conversation protocol $\mathcal{C}$ is* realizable *if there exists conversation protocols $\mathcal{C}_{p_0}, \dots, \mathcal{C}_{p_{|P|}}$ for each peer, such that the channel system $\dot{\mathcal{C}} = \mathcal{C}_{p_0} \otimes \dots \otimes \mathcal{C}_{p_{|P|}}$ is such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\dot{\mathcal{C}})$. $\mathcal{C}$ is* strictly realizable *when $\mathcal{C}_{p_i} = \pi_{p_i}(\mathcal{C})$ for $0 \leq i \leq |P|$.*

3. SHARED-STATE PROJECTIONS

A simple way of ensuring realizability is to develop the channel system $\dot{\mathcal{C}}$ and to exhaustively check that no spurious send trace is produced. However, the presence of unbounded message queues makes the problem in general intractable, even for simple protocols [11]. Other criteria must be developed to decide realizability of a conversation protocol. We propose *Shared-State Projections* (SSP) for this purpose.

3.1 An Intuition for SSPs

An SSP is a different kind of peer projection for a conversation protocol. Rather than simply projecting the protocol over a single peer p, an SSP also keeps track, in parallel, of the state of other peers —or more precisely, of what can be deduced about that state, based on messages that p sends and receives.

Consider the simple conversation protocol shown in Figure 4. Suppose, from A's point of view, that the message exchange has just started, i.e. all peers are in state 0. It shall first be noted that, from A's point of view, transitions from state 0 to state 3 and from state 3 to state 5 are invisible transitions; therefore, the current state of A is actually the set $\{0, 3, 5\}$. Moreover, from A's point of view, 0 is not the only possible state for peer B:

- Since A cannot distinguish between states 0, 3 and 5, B can reach these states with no way to know whether this transition was indeed taken.
- B cannot distinguish between states where it is neither the sender nor the receiver. Hence, 2 is (vacuously) a possible state.
- B cannot be in state 1, since it would require A sending m_1. For a similar reason, 6 is unreachable, as it relies on C's participation —which in turn depends on A.
- Finally, because communications are asynchronous, B can send a message to A without A having received it yet. Hence, when A is in state 0, before it reads anything from its input queue, B can already be in state 4.

From this informal chain of deductions, peer A in state 0 "knows" that peer B can be in states $\{0, 2, 3, 4, 5\}$, but *cannot* be in states $\{1, 6\}$. With a similar reasoning about C, A can guess that its possible states are $\{0, 1, 3, 4, 5\}$.

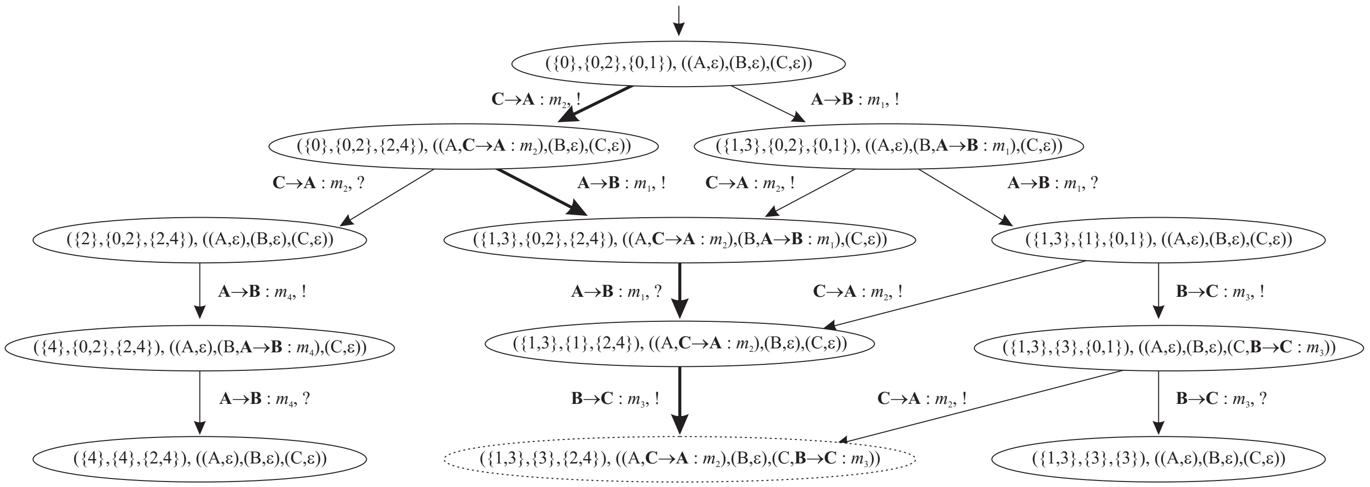

({0},{0,2},{0,1}), ((A,ε),(B,ε),(C,ε))
C→A : m2, !
A→B : m1, !
({0},{0,2},{2,4}), ((A,C→A : m2),(B,ε),(C,ε))
({1,3},{0,2},{0,1}), ((A,ε),(B,A→B : m1),(C,ε))
C→A : m2, ?
A→B : m1, !
C→A : m2, !
A→B : m1, ?
({2},{0,2},{2,4}), ((A,ε),(B,ε),(C,ε))
({1,3},{0,2},{2,4}), ((A,C→A : m2),(B,A→B : m1),(C,ε))
({1,3},{1},{0,1}), ((A,ε),(B,ε),(C,ε))
A→B : m4, !
A→B : m1, ?
C→A : m2, !
B→C : m3, !
({4},{0,2},{2,4}), ((A,ε),(B,A→B : m4),(C,ε))
({1,3},{1},{2,4}), ((A,C→A : m2),(B,ε),(C,ε))
({1,3},{3},{0,1}), ((A,ε),(B,ε),(C,B→C : m3))
A→B : m4, ?
B→C : m3, !
C→A : m2, !
B→C : m3, ?
({4},{4},{2,4}), ((A,ε),(B,ε),(C,ε))
({1,3},{3},{2,4}), ((A,C→A : m2),(B,ε),(C,B→C : m3))
({1,3},{3},{3}), ((A,ε),(B,ε),(C,ε))

Figure 3: A portion of a channel system obtained from the composition of the peer projections in Figure 2(b–d).

The initial state of A's SSP is therefore a *triplet* of sets of states, one for each peer, as follows: A: $\{0, 3, 5\}$, B: $\{0, 2, 3, 4, 5\}$, C: $\{0, 1, 3, 4, 5\}$. From that list of possible states, one can then compute possible *transitions*, leading to new combinations of possible states, and so on:

Definition 4. *Let $\mathcal{C} = (P, S, s_0, L, \delta)$ be a conversation protocol, with $|P|$ the number of peers involved in $\mathcal{C}$. A* shared-state projection *(SSP) derived from $\mathcal{C}$, from p's point of view, noted $\widehat{\pi}_p(\mathcal{C})$, is a quintuplet $(P, \widehat{S}, \widehat{s}_0, L, \widehat{\delta})$, where $\widehat{S} = (2^S)^{|P|}$ is a set of* shared peer states, *$\widehat{s}_0 \in \widehat{S}$ is the initial state, and $\widehat{\delta} : \widehat{S} \times P \times L \times P \rightarrow \widehat{S}$ is a transition function.*

An SSP can be seen as a "hybrid": like channel systems, one keeps track of states for all peers, but like conversation protocols, queue contents are not directly taken into account.

3.2 Conditions on SSPs

We now proceed to give conditions on an SSP that, taken together, define an algorithm for computing it. The informal presentation of an SSP showed that guessing possible states for other peers involves computing some kind of "reachability function" over a set of starting states. Indeed, the possible state of each peer is interdependent with the actions performed by other peers. For example, in Figure 5a, B cannot advance to state s_3 along the path $s_0 \ldots s_3$ without A first sending m_1 to D, and D then sending m_2 to B. In other words, A indirectly "controls" or "blocks" B from reaching s_3 along that path. On the contrary, in Figure 5b, A does not constrain B from reaching s_3. This notion is formalized below.

Definition 5 (Blocking peer). *Let $\bar{s} = s_1 s_2 \ldots s_n$ be a path in $\mathcal{C}$, and let p be some peer. The* peers blocked by p at s_n along π, *noted $B_p(\bar{s})$, is the smallest set of peers such that:*

1. *$p \in B_p(\bar{s})$*
2. *if there exists a transition $p_1 \xrightarrow{m} p_2$ from a pair of states s_i, s_{i+1} in $\bar{s}$, and $p_1 \in B_p(\bar{s})$, then $p_2 \in B_p(\bar{s})$.*

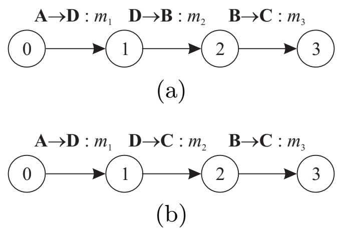


Figure 5: Different scenarios for blocked peers along a path

From this definition of a blocking peer, we develop a form of reachability, called (p, p')-*reachability*.

Definition 6 ((p, p')-reachability). *Let p, p' be two peers and $s, s' \in S$ be two states of a conversation. The state s' is (p, p')-*reachable *from s if and only if there exists a path $s_0, \ldots, s_n$ ($s_0 = s$, $s_n = s'$) and some $0 \leq k \leq n$ such that: 1) along $s_0 \ldots s_k$, the sender is not blocked by p', and along $s_k \ldots s_n$, p is neither the sender nor the recipient of any message.*

To get B's possible states in our motivating example, we computed all states that are (A,B)-reachable from state 0. We indeed obtain the set $\{0, 2, 3, 4, 5\}$. Similarly, possible states for A and C have been obtained through (A,A)- and (A,C)-reachability. Thus we obtain a formal definition of an initial state in an SSP:

Definition 7 (Initial state). *The initial state $\widehat{s}_0$ of an SSP $\widehat{\mathcal{C}}$ developed from p's point of view is the $|P|$-tuple $(\sigma_0, \ldots, \sigma_{|P|})$ such that σ_i is the set of states (p, p_i)-reachable from s_0, the initial state of $\mathcal{C}$.*

From this initial state of an SSP, one is then interested in finding possible *transitions*, and to compute the resulting state for each transition. We need to introduce two auxiliary functions that will be used in this computation. We first extend a transition function δ with $\delta^{\#}$, that returns a special symbol # standing for "empty". More precisely, $\delta^{\#}(s) = \delta(s)$ when $\delta(s)$ is defined, and $\delta^{\#}(s) = \#$ otherwise. Similarly, for a subset $S' \subseteq S$, we have $\delta^{\#}(S') = \delta(S')$ if $\delta(S') \neq \emptyset$, and $\delta^{\#}(S') = \#$ otherwise.

A second useful function that will be used is the $Post(S', \ell)$ function, which removes from S' all states that precede the first occurrence of some message label ℓ in the conversation protocol $\mathcal{C}$.

DEFINITION 8 (*Post* FUNCTION). *Let $S' \subseteq S$ be a subset of states of some conversation, and $\ell \in L$ be some message label. The function $Post(S', \ell)$ returns the smallest subset such that:*

1. *If there exists $p_i, p_j \in P$, $s \in S$ such that $\delta(s, p_i, \ell, p_j) = s'$, then $s' \in Post(S', \ell)$*
2. *If $s \in Post(S', \ell)$, $s' \in S'$ and there exists $p_i, p_j \in P$, $\ell' \in L$ such that $\delta(s, p_i, \ell, p_j) = s'$, then $s' \in Post(S', \ell)$.*

Finally, we require one last reachability definition, namely *?p-reachability*:

DEFINITION 9 (?p-REACHABILITY). *Let $\mathcal{C}$ be a conversation protocol, and $p \in P$ be some peer. A state s' is ?p-*reachable *from s if and only if there exists a path of messages from s to s' in $\mathcal{C}$ where p is never the sender.*

This intuitively corresponds to the fact that a peer p can be in some state s, and have messages waiting to be read in its input queue. Therefore, s' can be reached by simply waiting or consuming messages in the queue; the state of the conversation protocol does not require p to send any new message to reach that state. Note that this does not mean that s' *will* be the state eventually reached by p after reading messages in its queue; it only means that s' is one of the possible reachable states in that situation.

For some state $s \in S$, we will note the set of all states (p, p')-reachable from s as $\mathcal{R}_S^{(p,p')}(s)$, or simply $\mathcal{R}^{(p,p')}(s)$ when the context is clear. By extension, for some set $S' \subseteq S$, we define $\mathcal{R}_S^{(p,p')}(S') \equiv \bigcup_{s \in S'} \mathcal{R}_S^{(p,p')}(s)$. The function $\mathcal{R}$ induces a partition of the set of states S' into multiple *regions*; the set of such regions will be noted $\mathcal{P}^{(p,p')}(S')$. Similar functions can be defined with ?p-reachability.

Equipped with these definitions, we can define $\widehat{\delta}$, the transition relation of an SSP.

DEFINITION 10 (TRANSITION RELATION). *Let $\widehat{s} = (S_{p_0}, \dots, S_{p_{|P|}})$ and $\widehat{s}' = (S'_{p_0}, \dots, S'_{p_{|P|}})$ be two states of the SSP $\widehat{\pi}_{p^*}(\mathcal{C})$, for some peer $p^* \in P$. Then $\widehat{\delta}(\widehat{s}, p_i, \ell_k, p_j) = \widehat{s}'$ if and only if, for each peer p, one of the following conditions holds:*

1. *$p \neq p_i$, $p^* \neq p_j$, and $S'_p = S_p$*
2. *$p = p_i$, and $S'_p = \mathcal{R}_S^{(p^*,p)}(\delta(S_p))$*
3. *$p^* \neq p_i$, $p \neq p_i$, $p \neq p_j$, and $S'_p = \mathcal{R}_S^{(p^*,p)}(Post(\widehat{s}_p, \ell_k))$*
4. *otherwise, $S'_p = \bigcup_{R \in \mathcal{P}^{?p}(S_p)} \delta^{\#}(R)$*

To build an SSP for some focus peer p^*, it suffices to create its initial peer state, and to recursively generate successors to each new peer state according to Definition 10. In each node of a shared-state projection, there are $|P|$ subsets of the original conversation protocol state set S, one for each peer. Therefore, an SSCS has at most $|P| \cdot 2^{|S|}$ distinct nodes, and the analysis requires computing at most n such projections (one for each peer). Therefore, the procedure is guaranteed to terminate.

3.3 A Realizability Condition

A first observation is that, from Definition 10, $\widehat{\pi}$ is an over-approximation of π, i.e. $\mathcal{L}(\pi_p(\mathcal{C})) \subseteq \mathcal{L}(\widehat{\pi}_p(\mathcal{C}))$ for every peer p. The next theorem follows immediately:

THEOREM 2. *Let $\mathcal{C}$ be a conversation, $\pi_{p_i}(\mathcal{C})$ ($0 \leq i \leq |P|$) and $\widehat{\pi}_{p_i}(\mathcal{C})$ be its peer projections and shared-state projections, respectively. Let $\dot{\mathcal{C}}$ be the channel system obtained from composing the π_{p_i}, and $\dot{\widehat{\mathcal{C}}}$ be the channel system obtained from composing the $\widehat{\pi}_{p_i}$, called a* shared-state channel system. *Then $\mathcal{L}(\dot{\mathcal{C}}) \subseteq \mathcal{L}(\dot{\widehat{\mathcal{C}}})$.*

From this result, we can immediately deduce that an SSCS is a sound approximation of realizability for conversations: any spurious message sequence produced by the asynchronous channel system, making the conversation protocol unrealizable, will also be a spurious message sequence of the SSCS. It remains to demonstrate some condition on an SSCS that could, in turn, warn of the presence of such a spurious message sequence. To this end, we develop the notion of a conflict state.

DEFINITION 11 (CONFLICT STATE). *Let $\widehat{s} = (S_{p_0}, \dots, S_{p_{|P|}})$ be a state of $\widehat{\pi}_{p^*}(\mathcal{C})$ for some peer p^*. Define $A(\widehat{s}) = S_{p_0} \cap \dots \cap S_{p_{|P|}}$. If $A(\widehat{s})$ is empty or contains the symbol $\#$, then $\widehat{s}$ is called a* conflict state. *Otherwise, the states in $A(\widehat{s})$ are called* possible agreements *for all peers.*

For example, with the conversation protocol in Figure 4 and the initial state $\widehat{s}$ for $\widehat{\pi}_A$ defined previously, we have $A = \{0, 3, 5\}$, the intersection of possible states for each peer. Intuitively, a conflict state describes a situation where, after some messages have been exchanged between peers, some of them “disagree” about the global state of this conversation, in a way that cannot be settled unless one of the peers sends a message or discards a symbol in the input queue —hence, as it is, the current state of each peer makes one of them “stuck”.

THEOREM 3. *Let $((S_{p_0}, \dots, S_{p_{|P|}}), w)$ be a global state of a shared-state channel system, and $A_{p_i} = A(S_{p_i})$ be the possible agreements for each of the SSPs in that global state. Then, unless one of the A_{p_i} is a conflict state, $A = A_{p_0} \cap \dots \cap A_{p_{|P|}} \neq \emptyset$.*

PROOF. By induction, we actually show that the intersection of the A_{p_i} must include the state of $\mathcal{C}$ resulting from the last send action. Base case: the initial state of $\dot{\mathcal{C}}$ vacuously follows the property. Induction step: suppose the property is true for any state $\dot{s}$ reachable within k steps from the initial state. Let $(\dot{s}, ((p_j, \ell, p_k), \star), \dot{s}')$ be a transition going one step further. Two cases must be considered:

1. If $\star =?$, since $\dot{s}$ contains an agreement, by the induction hypothesis all paths from $\dot{s}$ taking “?” transitions have the same agreement; this includes all paths going through $\dot{s}'$.
2. If $\star =!$, let s_1 be the agreement in $\dot{s}$. Let $s_2 = \delta(s_1, p_j, \ell, p_k)$; let us show that if an agreement can be reached, it has to include s_2.

   For p_j, by Definition 10.2, s_2 is already in its set of states. Since p_k could agree on s_1 in $\dot{s}$, it suffices for

p_k to reach s_1 through receive transitions, and then to consume (p_j, ℓ, p_k); this is possible by Definition 10.4. Then, by construction, s_2 is an element of S_k in $\dot{s}'$. All other peers are neither the recipients nor the senders; for them, the transition from s_1 to s_2 is an ϵ-transition, and therefore s_2 is already in their set of states by Definition 10.3. Hence all peers can reach an agreement on s_2 without sending any new message.

□

A consequence of Theorem 3 is that, if none of the SSP contains a conflict state, then neither does the channel system built from them. Conversely, the presence of a conflict state might hint at a spurious message sequence.

THEOREM 4. *Let $\hat{\dot{\mathcal{C}}}$ be the SSCS obtained from the SSPs of some conversation protocol $\mathcal{C}$. Let $\overline{\dot{s}} = \dot{s}_0, \dot{s}_1, \ldots$ be a state trace of $\hat{\dot{\mathcal{C}}}$ where no state is a conflict state. Then the send trace of $\overline{\dot{s}}$ is a trace of $\mathcal{C}$.*

PROOF. For a global state $\dot{s}$, define A as in the previous theorem. From $\overline{\dot{s}}$, we build a sequences of states of $\mathcal{C}$: $\overline{s} = A(\dot{s}_0), A(\dot{s}_1), \ldots$. By Theorem 3, $A(\dot{s}_i)$ is well-defined and changes only through send operations. Therefore, for $\dot{s}_i, \dot{s}_{i+1}$ two successive states:

- if the transition results from a message $((p, \ell, p'), ?)$, then $A(\dot{s}_{i+1}) = A(\dot{s}_i)$;
- if the transition results from an action $((p, \ell, p'), !)$, then $\sigma(\dot{s}_{i+1}) = \delta(A(\dot{s}_i), ((p, \ell, p'), !))$.

It follows that $\overline{\dot{s}}$'s send trace is a trace of $\mathcal{C}$. □

A realizability condition follows from all these results:

COROLLARY 1. *Let $\mathcal{C}$ be a conversation. If none of the SSPs $\hat{\pi}_{p_0}(\mathcal{C})$, …, $\hat{\pi}_{p_{|P|}}(\mathcal{C})$ contain a conflict state, then $\mathcal{C}$ is realizable.*

PROOF. Let $\dot{\mathcal{C}}$ be the channel system obtained by composing the peer projections $\pi_{p_i}(\mathcal{C})$ for each peer p_i. Let $\hat{\dot{\mathcal{C}}}$ be the SSCS obtained by composing the SSPs $\hat{\pi}_{p_i}(\mathcal{C})$ for each peer. By theorem 1, $\mathcal{L}(\mathcal{C}) \subseteq \mathcal{L}(\dot{\mathcal{C}})$. By theorem 2, $\mathcal{L}(\dot{\mathcal{C}}) \subseteq \mathcal{L}(\hat{\dot{\mathcal{C}}})$. By theorem 3, if none of the SSPs contain a conflict state, then same holds for $\hat{\dot{\mathcal{C}}}$. Finally, by theorem 4, in such a case $\mathcal{L}(\hat{\dot{\mathcal{C}}}) \subseteq \mathcal{L}(\mathcal{C})$. It follows that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\dot{\mathcal{C}}) = \mathcal{L}(\hat{\dot{\mathcal{C}}})$, and hence $\mathcal{C}$ is (strictly) realizable. □

Corollary 1 provides a *sufficient* condition for a conversation protocol $\mathcal{C}$ to be realizable. Moreover, this condition can be computed by reachability analysis. Indeed, we need to compute $|P|$ SSPs, where each has a finite size bounded by $|P| \cdot 2^{|S|}$. This gives an upper bound of $|P|^2 \cdot 2^{|S|}$ on the size of the state space, thereby reducing the problem to a search in a finite graph. We shall see in a later experimental evaluation of the approach that this upper bound is seldom reached for real-world protocols.

4. EXPERIMENTAL RESULTS

To test our approach, we developed a tool called SSPCalc, which takes as input a conversation protocol, and then computes SSPs for some or all of the peers involved. The tool is implemented in PHP and runs on a web server; it uses a graphical interface that allows a user to specify various computation options. SSPCalc's results are also displayed graphically. The channel system corresponding to each SSCS, as well as the resulting product, are all represented as annotated graphs. Various statistics about the computation, such as memory consumption and state space size, are also given.

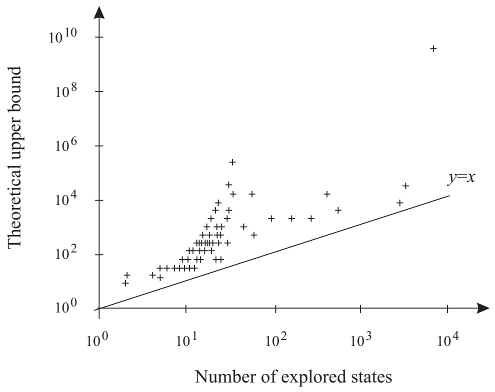


Figure 6: Actual state space size vs. theoretical upper bound for each protocol.

Analysis of Performance: We ran SSPCalc on more than 100 different real-world protocols, taken from the following sources: IBM Conversation Support [1], Channel contracts from the Singularity operating system [20], protocols from the BPEL Specification [7], and the travel agency example from [17]. All of them were already specified graphically as conversation protocols; they simply had to be translated into an equivalent textual representation that SSPCalc uses as input.

We first computed the total state space that was required to be explored for each protocol. We compared this number with the theoretical $|P|^2 \cdot 2^{|S|}$ upper bound on the number of states. These results are shown in Figure 6. As one can see, the actual number of states explored in every protocol is much lower than the theoretical maximum. The log-log graph has a slope of 2.5, which means that the expected number of reachable states is roughly in a $\sqrt{x^5}$ relationship with the upper bound.

Although SSPCalc is a proof-of-concept prototype written as a PHP script, we still measured the total time for the validation of each protocol, in terms of its state space size, and plotted the results in Figure 7. The log-log graph has a slope of 1.75, which indicates that processing time is roughly the square of the state space size. The times ranged from 8 milliseconds to one hour, depending on the protocol. Very few protocols took a long time to process: 91% of them were analyzed in less than one second, and 95% in less than 10 seconds. They were measured on an Intel machine running at 2.53 GHz under Linux CentOS.

False Positives: More importantly, we compared the results returned by SSPCalc with those returned by WSAT [9], a tool for analyzing conversation protocols, and Tune [20], a tool for analyzing Singularity channel contracts. WSAT uses the three realizability conditions defined in [10, 11] called synchronous-compatible, autonomous, and lossless-join (de-

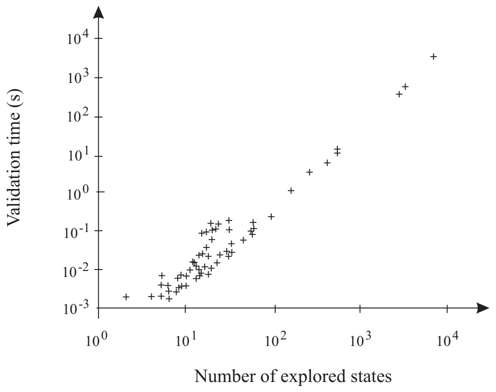


Figure 7: Total verification time with respect to state space size.

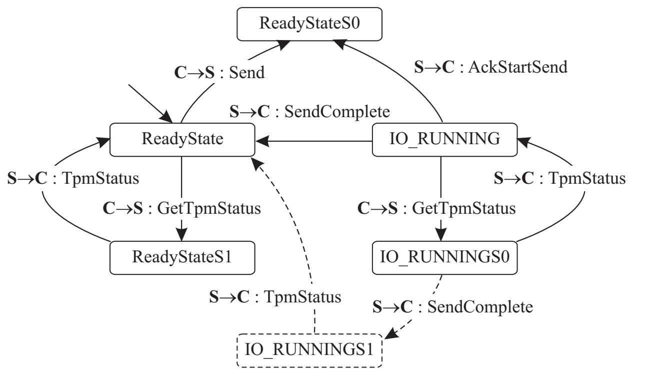


Figure 8: The original TPMContract protocol (solid lines), with additional states and transitions (dashed) to make it realizable.

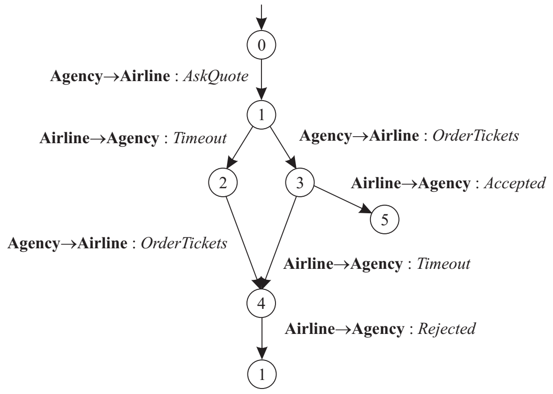
scribed in Section 5 below) whereas Tune uses only the autonomous condition (which is sufficient for determining realizability of two-party protocols like channel contracts).

Both WSAT and Tune use sound realizability analyses and, hence, correctly identify the protocols that are unrealizable. For example, the HAGGLE protocol from IBM Conversation Support [1], and the TPMContract protocol, shown in Figure 8, are two unrealizable protocols that are correctly identified as such by WSAT and Tune. SSPCalc is also a sound realizability analysis tool, so it also identifies these two protocols as unrealizable.

In a way similar to the example of Figure 1, it is possible to fix some unrealizable protocols with additional transitions and states, in such a way that it becomes realizable. A fixed version of TPMContract, provided in [20], is shown as dashed states and transitions in Figure 8. However, this fixed version still violates the autonomous condition and hence fail the realizability tests used in WSAT and Tune.

Since WSAT and Tune use sound, but incomplete analyses, this does not mean that the protocol is unrealizable. In fact, SSPCalc computed all SSPs for the fixed protocol and found no conflict state. By Corollary 1, this is sufficient to conclude that the fixed version of TPMContract is realizable, a result that cannot be shown by the realizability analyses used in WSAT and Tune.

Another example that causes the realizability conditions used in WSAT and Tune to generate false positives is the



0
Agency→Airline : AskQuote
1
Airline→Agency : Timeout
Agency→Airline : OrderTickets
2
3
Airline→Agency : Accepted
5
Agency→Airline : OrderTickets
Airline→Agency : Timeout
4
Airline→Agency : Rejected
1

Figure 9: A portion of the fixed airline protocol from [17]

airline example mentioned in [17], where the original protocol deadlocks because of a timeout. Since this timeout can be sent while the agency is sending an order, to fix it, the protocol must be modified such that both orderings of timeout and order are allowed from a global point of view. The fixed protocol is shown in Figure 9.

Based on the realizability conditions used by WSAT and Tune, this fixed protocol is identified as unrealizable. However, SSPCalc correctly identifies the fixed protocol in Figure 9 as realizable. Even for the fixed version of the simple file transfer protocol shown in Figure 1b, WSAT and Tune generate false positives and report that it is unrealizable, whereas SSPCalc correctly identifies that it is realizable.

The protocols we discussed above, for which the WSAT and Tune tools report false positives, can be characterized as *arbitrary-initiator* protocols:

DEFINITION 12. *A conversation protocol $\mathcal{C}$ is an* protocol with arbitrary-initiator *if there exists some state $s \in S$, peers p_i, p_j, p_m, p_n and message labels ℓ, ℓ' such that both $\delta(s, p_i, \ell, p_j)$ and $\delta(s, p_m, \ell', p_n)$ are defined, and $p_i \neq p_m$.*

Such protocols can be unrealizable, since, in the same state, two different peers can decide to send a message, without "warning" the other of its intent. Yet, as we have shown above, not every such protocol is unrealizable: in fact, any conversation protocol where some peer can interrupt a sequence of messages sent by another peer exhibits this arbitrary-initiator property. This is the case for the "fixed" version of the file transfer protocol, shown in Figure 1b. This conversation protocol accepts both sequences s, c, f and s, f, c, which is exactly what its corresponding channel system produces. Therefore, it is realizable. Since the initiator of the protocol cannot be guessed by any of the peers, the specification compensates by providing an appropriate behavior for both orderings. In the case of the file transfer protocol, this would simply mean that the server accepts, but ignores any cancellation messages received after the transfer is finished.

5. RELATED WORK AND DISCUSSION

The present work can be compared to earlier results on assessing the realizability of conversations, with a special focus on their behavior on arbitrary-initiator protocols.

Local enforceability studied in [23] is concerned with finding individual specifications for each communicating peer such that their composition produces a predefined set of interactions, expressed in a web service *choreography language*. However, in this approach synchronous communication is used, and hence the arbitrary initiator problem does not occur.

Message patterns expressed with Petri nets using synchronous communication are "de-synchronized" in [6] —that is, one is interested in finding a specification that produces the same pattern of messages when communications become asynchronous. This work, however, already assumes that a conversation protocol is realizable and does not provide realizability conditions.

Realizability Conditions for Conversation Protocols: In [10] a set of three conditions for conversation protocols is presented, which, taken together, are *sufficient* to ensure realizability:

1. Synchronous-compatible: A conversation protocol $\mathcal{C}$ is synchronous-compatible if every time a peer p has a send transition for some label ℓ to some peer p', then p' has a receive transition for that same label, or should reach such a transition through ϵ-transitions.

2. Autonomous: A conversation protocol $\mathcal{C}$ is autonomous if each peer, at any moment, can do only one of the following: 1) terminate, 2) send a message, or 3) receive a message.

3. Lossless-join: If we let $\mathcal{C}' = \pi_{p_0}(\mathcal{C}) \times \cdots \times \pi_{p_{|P|}}(\mathcal{C})$ be the conversation protocol obtained as the "Cartesian product" of each peer projection, a conversation protocol $\mathcal{C}$ is lossless join if $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{C}')$.

Note that the lossless join condition operates on a product of peer projections (which is still a conversation protocol), and not on the channel system built from these projections. Otherwise, the lossless join condition would be equivalent to strict realizability. Algorithms for checking the above conditions on conversation protocols have been shown in [11]. The WSAT [9] and Tune [20] tools we mentioned above implement these algorithms.

While the above conditions allow a sound analysis of many protocols, the major drawback comes from the autonomous condition. In fact, the autonomous condition is the exact negation of the definition of an arbitrary-initiator protocol. It follows that this method cannot discriminate between realizable and unrealizable classes of such protocols, and immediately labels them as unrealizable. Hence, as we discussed above, WSAT and Tune generate false positives for realizable arbitrary initiator protocols as expected.

Session Types: In session types conformance of an interaction with a predefined protocol is analyzed as a typing problem. Intuitively, if T_1, T_2 are two types, then a type T_3 can be built as $T_1.T_2$, where the period denotes the sequencing operator. A global interaction $\overline{m}$ between peers (called a *session*) will conform to type T_3 if it can be divided into a sequence of two interactions, $\overline{m}_1.\overline{m}_2$, such that $\overline{m}_1$ conforms to type T_1, and $\overline{m}_2$ conforms to type T_2. Similar rules can be defined for non-deterministic branching, conditional branching, parallel execution.

The approach in session types [13], is to first devise a global type, G, to an intended scenario consisting of interacting peers. Different programmers can independently develop portions of this interaction by taking the projection of G on each of the peers. If the individual program for each peer p can be shown to conform to its respective projection of G (noted $G \upharpoonright p$), i.e., if each program is "typable", then the rules of session types ensure that when the programs are executed, they follow the stipulated global scenario. One can see the immediate parallel between the rationale behind session types and the definition of strict realizability presented in Section 2.3. More precisely, realizability of interactions can be ensured through a "type discipline" by checking two conditions:

1. A programmer first describes a scenario as a global type G and checks that it conforms to the so called *linearity* condition.

2. Each component of the interaction is then developed independently, and periodically checked to make sure it is *typable* against the projection of G.

If linearity of G and typability of each program is enforced, realizability follows. Session types can therefore provide an alternate, sound analysis to the realizability conditions described above.

It turns out, however, that the typing system for session types contains an analogue of the autonomous condition shown above. More precisely, rule "If" in the type discipline shown in [12] is formulated as follows:

$$\frac{\Gamma \vdash e \triangleright \mathbf{bool} \quad \Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \text{if } e \text{ then } P \text{ else } Q \triangleright \Delta}$$

Intuitively, this rule shows that for a branch of the form "if e then P else Q" to conform to type Δ (bottom part of the rule), then e must be of Boolean type, and both branches P and Q must conform to the same type Δ. Since a type determines the sender and recipient of some communication event, it follows that P and Q must have the same sender and recipient, no matter what e is.

Similar restrictions occur for other forms of branching in the type system. For example, the construct $k \triangleright \{l_1 : P_1 [] \cdots [] \; l_n : P_n\}$, called *label branching* in [12], represents a peer offering, over some communication channel k, multiple choices $P_1, \ldots, P_n$ for the continuation of a session. The P_i need not be of the same type, and hence nothing prevents a peer from being alternately a sender and a receiver in some of these choices. However, for such an expression to be typable, some other peer must include as its *cotype* the expression $k \triangleleft l; P$, where the actual choice of label is communicated over k. This is shown in Figure 10; one can see that this no longer represents an arbitrary-initiator protocol, since the communication of l_i over channel k is always unidirectional. Basically, the session type discipline prevents well-typed programs from including arbitrary-initiator branching. Hence, the realizability of such protocols cannot be established using the type rules of session types.

6. CONCLUSION

The shared-state projections (SSP) presented in this paper provide a novel and sound approach to determining realizability of message-based interactions. In contrast to earlier work, SSP is based on the intuitive notion that a realizable protocol is a protocol where each peer, at any moment, should have a "clear enough" indication of the current state of other peers. By computing, for each state and from each peer's point of view, the possible states for other peers in

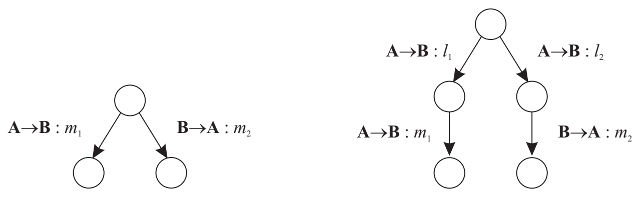


Figure 10: Arbitrary-initiator protocol (left) and label branching in session types (right).

a conversation, we have shown that unrealizable protocols can be soundly identified through the identification of "disagreements" between two peers' global states. The realizability check we developed based on SSP correctly identifies the realizability of a class of protocols called arbitrary-initiator protocols, whereas earlier realizability conditions always produce false positives for such protocols.

An interesting feature of the SSP approach is that its output is not simply Boolean. While the shared-state projections are used to derive a verdict about realizability, these models in themselves can be put to other uses. As we have shown, computing the Cartesian product of each shared-state projection yields an over-approximation of the original specification. By removing conflict nodes from this product, one obtains a subset of the conversation protocol which is guaranteed to be realizable, at no additional cost. Conversely, by adding transitions to the original conversation, one can convert conflict nodes into "correct" nodes until no conflict remains; the augmented protocol can be seen as a "fixed" version of the original. Therefore, we believe that it would be possible to extend the SSP approach for automatically *repairing* unrealizable protocols.

7. REFERENCES

[1] IBM conversation support project, 2002. http://www.research.ibm.com/convsupport.

[2] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

[3] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *13th International Symposium on Distributed Computing (DISC)*, pages 1–18, 1999.

[4] T. Bultan, X. Fu, and J. Su. Analyzing conversations: Realizability, synchronizability, and verification. In L. Baresi and E. D. Nitto, editors, *Test and Analysis of Web Services*, pages 57–86. Springer, 2007.

[5] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming, W3C note, October 2006. http://www.w3.org/2002/-ws/chor/edcopies/theory/note.pdf.

[6] G. Decker, A. P. Barros, F. M. Kraft, and N. Lohmann. Non-desynchronizable service choreographies. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 331–346, 2008.

[7] D. Jordan et al. Web services business process execution language version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[8] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. 2006 EuroSys Conf.*, pages 177–190, 2006.

[9] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Service Compositions. In *Proc. 16th Int. Conf. on Computer Aided Verification*, pages 510–514, 2004.

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

[11] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

[12] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symp. on Programming on Programming Languages and Systems (ESOP'98)*, pages 122–138, 1998.

[13] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *POPL*, pages 273–284. ACM, 2008.

[14] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

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

[16] R. Kazhamiakin and M. Pistore. Analysis of realizability conditions for web service choreographies. In *FORTE*, pages 61–76, 2006.

[17] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and participant synthesis. In M. Dumas and R. Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2007.

[18] D. A. Menascé. MOM vs. RPC: Communication models for distributed applications. *IEEE Internet Computing*, 9(2):90–93, 2005.

[19] Microsoft Message Queuing Service. http://www.microsoft.com/windowsserver2003 /technologies/msmq/default.mspx.

[20] Z. Stengel and T. Bultan. Analyzing Singularity channel contracts. In G. Rothermel and L. K. Dillon, editors, *ISSTA*, pages 13–24. ACM, 2009.

[21] Web services reliable messaging. http://www.ibm.com/developerworks/library /specification/ws-rm/.

[22] Web Service Choreography Description Language (WS-CDL). `http://www.w3.org/TR/ws-cdl-10/`, 2005.

[23] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker. Service interaction modeling: Bridging global and local views. In *EDOC*, pages 45–55. IEEE Computer Society, 2006.