Verification of Communicating Data-Driven Web Services

Alin Deutsch Liying Sui Victor Vianu Dayou Zhou

University of California, San Diego
Computer Science and Engineering
{deutsch,lsui,vianu,dzhou}@cs.ucsd.edu

ABSTRACT

We study the verification of compositions of Web Service peers which interact asynchronously by exchanging messages. Each peer has access to a local database and reacts to user input and incoming messages by performing various actions and sending messages. The reaction is described by queries over the database, internal state, user input and received messages. We consider two formalisms for specification of correctness properties of compositions, namely Linear Temporal First-Order Logic and Conversation Protocols. For both formalisms, we map the boundaries of verification decidability, showing that they include expressive classes of compositions and properties. We also address modular verification, in which the correctness of a composition is predicated on the properties of its environment.

1. INTRODUCTION

Recent years have witnessed the proliferation of Web services powered by an underlying database and interacting with human users and with peer Web services. Examples include e-commerce sites, scientific and other domain-specific portals, e-government, and data-driven Web services. The development of such services is facilitated by the emergence of high-level specification tools which automatically generate the code implementing the Web service (a commercially successful representative is WebML [7]). Besides increasing developer productivity, high-level specification tools create opportunities for automatic verification. Such verification leads to increased confidence in the service's correctness by addressing the most likely source of errors –the specification itself– as opposed to the less likely errors in the well-maintained automatic code generator.

In prior work [13], we studied as a first step the verification of isolated Web services which interact only with external users (through a Web browser interface). Many settings however require services to interact with each other, typically by exchanging messages. For instance, even seemingly self-contained e-commerce Web sites place calls to an external Web service to charge a credit card. Similarly, a bank's loan management application exchanges messages with a credit reporting agency's Web service.

In this paper, we present a significant extension of our verification work to compositions of Web services (also called *peers*), which interact by asynchronous message exchange. The peers receive both input from their users (through the Web interface) and messages from other peers. They react by updating their internal state, by sending messages (such as a credit check request to a credit agency's Web service)

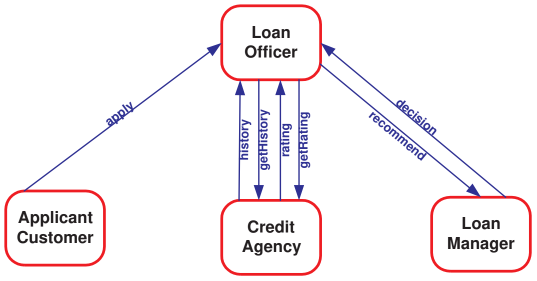

Figure 1: Bank loan application

or by performing actions (such as the generation of a notification letter). Each peer's reaction is a function of the current contents of the database, state, user input and received messages. We illustrate a composition below.

Example **1.1** Consider a bank's loan application process involving the applicant customer, the loan officer, his manager, and the credit reporting agency (shown in Figure 1). Applicant, loan officer and manager play their part in the process using Web interfaces running on top of the Web services $\mathcal{A}, \mathcal{O}, \mathcal{M}$ respectively. The credit reporting agency provides the Web service $\mathcal{CR}$.

The officer's Web service $\mathcal{O}$, for instance,

- receives application messages from the customer's Web service;
- allows the officer to view details about the applicant, available in the bank's customer database;
- allows the officer to request the customer's credit history, obtained via a message from $\mathcal{CR}$, which retrieves the information by querying its own local database;
- allows the officer to input his recommendation of acceptance or denial;
- sends a message with the recommendation, as well as the customer data and credit history to the Web service $\mathcal{M}$, which allows the manager to input her final decision, returned as a message to $\mathcal{O}$;
- generates notification letters for customers.

As a sample correctness property, we'd like to ensure that the composition satisfies bank policy, according to which the officer may make an unsupervised decision granting loans to applicants with excellent credit rating and denying them to those with poor rating. All other credit ratings require the manager's involvement. We show in this paper how such properties are expressed and automatically verified. □

A *run* is a sequence of snapshots through which the collection of peers evolves during the interaction with users and with each other. The correctness of a composition is specified by properties which express requirements on individual run snapshots (using First-Order Logic), as well as the temporal relationship among these snapshots. Verification involves searching for runs which violate the property. We are interested in *sound and complete* verification, which produces a counterexample run if and only if the property is violated.

We consider two formalisms for property specification, namely First-Order Linear Temporal Logic and Conversation Protocols. Conversation protocols were introduced in [17] as a generalization of an industrial standard (IBM's conversation support project [19]). Classical conversation protocols are concerned only with the sequence of message names observed during the interaction. In this paper, we extend them with awareness of the message contents.

Contributions. For both property formalisms, we map the boundaries of verification decidability. In particular, we explore various semantics for message-based communication (singleton versus set messages, lossy versus perfect communication channels, bounded versus unbounded received message queues). We also identify syntactic restrictions on the peer and property specifications which, under appropriate communication semantics, guarantee decidability of verification in PSPACE. This complexity is the best one can hope for given that propositional LTL verification of finite-state Mealy machines is PSPACE-complete [9]. We show that our restrictions are quite tight: even slight relaxations thereof lead to undecidability. When the composition consists of a single peer with no message channels, the restrictions degenerate to the notion of *input-boundedness* from [13]. We demonstrated the expressivity of input-bounded peer specifications in [12] by modeling significant parts of four well-known database-powered Web sites (demo available at [1]). The favorable experimental results obtained in [12] for verification of individual input-bounded services suggest that similarly good performance can be expected for compositions.

Finally, we address modular verification, in which the correctness of a subset of the peers is checked when the full specification of the other peers is not available and the only knowledge about them is declared as properties of their message input-output behavior. Modular verification is useful when some peers are provided by autonomous parties unwilling to disclose implementation details, or when verification of a partially specified composition is desired.

Relationship to Software Verification. In the broader context of software verification, our work addresses sound and complete automatic verification of a novel class of reactive systems communicating asynchronously. The systems are infinite-state because the underlying database and user input values are not fixed in advance. This is a departure from most existing research, which focuses on communicating finite-state systems (called CFSMs in [6, 2, 3], and *e-compositions* in the context of Web services, as surveyed in [22, 23, 24, 21]). Conventional wisdom in software verification holds that sound and complete verification of infinite-state systems is infeasible, prescribing instead the approach of finite-state abstraction followed by classical finite-state model checking. In the data-driven Web service scenario we consider, data values are first-class citizens and abstracting them away is not satisfactory. For instance, abstraction would allow us to check that upon receiving *some* credit score request, the reporting agency sends *some* reply message, but preclude us from requiring the reply to reflect the customer's database record. To handle data-driven compositions and data-aware correctness properties, we employ a novel mix of model-checking and database/logic techniques. Our results suggest that the data-driven composition scenario with peer specifications based on database queries is particularly well-suited to automatic verification, in contrast to general-purpose software verification.

Paper Outline. The remainder of the paper is organized as follows. Section 2 introduces our formalism for specification of data-driven peers and compositions. Sections 3 and 4 study the verification of properties expressed by Linear Temporal First-Order Logic, respectively Conversation Protocols. We address modular verification in Section 5. We present related work in Section 6 and conclude in Section 7. All proofs are shown in Appendix A.

2. PEERS AND COMPOSITIONS

We describe a framework for the specification of compositions, starting from the individual Web services (called peers) involved in a composition. Peers communicate with each other by sending and receiving messages via one-way channels implemented by *message queues*. Each queue is associated with a unique sender who places messages into the queue, and a unique receiver who consumes messages from it in FIFO order. The queue is called an *out-queue* by the sender and an *in-queue* by the receiver. The queues are classified into *flat queues* and *nested queues*. Flat queues deliver single-tuple messages, e.g. the age and social security number of a given customer. Nested queues transport messages consisting of a set of tuples, e.g. the set of books written by an author. Notice that, by modeling communication channels with queues, we assume that messages arrive in the same order they were sent.

Each peer consists of

–an underlying database that remains fixed throughout the interaction with the environment;[1]

–a set of state relations that are updated throughout the interaction;

–a set of input relations which capture the input provided by the user who picks among a set of options generated by the peer.

–a set of action relations modeling the performed actions (e.g. the sending of a notification letter is modeled as the insertion of a tuple into the letter table)

–a set of in-queues through which the Web service receives messages;

–a set of out-queues used to send messages;

–a set of rules specifying the reaction to user input and received messages.

The rules specify how the set of current user input choices is generated, and how the Web service reacts to the user's input and/or to the arrival of messages. The reaction is a function of the current contents of the database, state, user input and received messages, and it involves updating the state, performing actions (such as the placement of an order) and sending messages (such as a credit check request to a

[1]We do not claim that the peer's database does not change; we simply regard the changing part of the database as state.

credit agency's Web service). Formally, we have:

Definition 2.1. *A* Web service *(or* peer*)* $\mathcal{W}$ *is a tuple* $\langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{Q_{in}}, \mathbf{Q_{out}}, \mathcal{R}\rangle$*, where:*

- $\mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{Q_{in}}, \mathbf{Q_{out}}$ *are relational schemas called, respectively, database, state, input, action, in-queue and out-queue schemas. The sets of relation symbols of the schemas are disjoint (but they may share constant symbols). The queue schemas are partitioned into a flat and a nested part:* $\mathbf{Q_{in}} = \mathbf{Q^f_{in}} \cup \mathbf{Q^n_{in}}$ *and* $\mathbf{Q_{out}} = \mathbf{Q^f_{out}} \cup \mathbf{Q^n_{out}}$.

We require $\mathbf{S}$ *to include, for each in-queue* Q*, a propositional state* $empty_Q$ *indicating if the queue* Q *is empty, referred to as a* queue state*. (Note that queue states are only available for in-queues, since we assume out-queues are located at the recipients of sent messages and thus their state is not accessible by the peer.) We also denote by* $\mathbf{Prev_I}$ *the relational vocabulary* $\{prev_I \mid I \in \mathbf{I}\}$*, where* $prev_I$ *has the same arity as* I *(intuitively,* $prev_I$ *refers to the most recent non-empty input to* I*).*

Finally, $\mathcal{R}$ *is a set of rules containing the following:*

- *For each input relation* $I \in \mathbf{I}$ *of arity* $k > 0$*, an* input rule
$$\text{Options}_I(\bar{x}) \leftarrow \varphi_I(\bar{x})$$
where Options_I *is a relation of arity* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi_I(\bar{x})$ *is an FO formula over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{Q_{in}}$*, with free variables* $\bar{x}$*.*
- *For each state relation* $S \in \mathbf{S}$ *that is not a queue state, one, both, or none of the following* state rules*:*
  - *an insertion rule* $S(\bar{x}) \leftarrow \varphi^+_S(\bar{x})$*,*
  - *a deletion rule* $\neg\, S(\bar{x}) \leftarrow \varphi^-_S(\bar{x})$*,*

  where the arity of S *is* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi^+_S(\bar{x})$*,* $\varphi^-_S(\bar{x})$ *are FO formulas over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{Prev_I} \cup \mathbf{Q_{in}}$*, with free variables* $\bar{x}$*.*
- *For each action relation* $A \in \mathbf{A}$*, an* action rule
$$A(\bar{x}) \leftarrow \varphi_A(\bar{x})$$
where the arity of A *is* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi_A(\bar{x})$ *is an FO formula over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{Prev_I} \cup \mathbf{Q_{in}}$*, with free variables* $\bar{x}$*.*
- *For each out-queue relation* $Q \in \mathbf{Q_{out}}$*, one* send rule
$$Q(\bar{x}) \leftarrow \varphi_Q(\bar{x})$$
where the arity of Q *is* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi_Q(\bar{x})$ *is an FO formula over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{Prev_I} \cup \mathbf{Q_{in}}$*, with free variables* $\bar{x}$*.*

Intuitively, the input rules specify a set of options to be presented to users, from which they can pick at most one tuple to input. This feature corresponds to menus in user interfaces. At every point in time, input J contains the current input tuple and $prev_J$ contains the most recent previous non-empty input to J. The state rules specify the tuples to be inserted or deleted from state relations (with conflicts given no-op semantics, as seen below). If no rule is specified for a given state relation, the state remains unchanged. The action rules specify the actions to be taken in response to the input. The send rules specify the tuples used to construct the message. For nested queues, all tuples yielded by one firing of the send rule are collected into one message. Flat queues are intended to be used when the send rule is known to yield a single tuple. If several tuples are generated, then at most one of them is non-deterministically placed into the queue and the others are dropped.

Remark. *Syntactic sugar* The above allows us to simulate various syntactic sugar on individual peers as done in [13] (such as a notion of a current Web page; rules governing the transition to the next page; inputs for buttons and HTML links; etc.). For simplicity, we abstract syntactic sugar and focus on modeling the distributed communication aspect of the composition.

Notation To improve readability, in the following we display any relation R in the specification of service $\mathcal{W}$, depending on whether it belongs to $\mathcal{W}.\mathbf{I}, \mathcal{W}.\mathbf{D}, \mathcal{W}.\mathbf{S}, \mathcal{W}.\mathbf{A}, \mathcal{W}.\mathbf{Q_{in}}$ and $\mathcal{W}.\mathbf{Q_{out}}$, as R, $\underline{\mathsf{R}}$, R, R, $?\mathbf{R}$ and $!\mathbf{R}$, respectively.

Example **2.2** We specify the loan officer's peer $\mathcal{O}$ from Example 1.1. $\mathcal{O}$'s schema is given as:

$$\begin{array}{rcl}
\mathcal{O}.\mathbf{D} & = & \{\underline{\mathsf{customer}}(cId, ssn, name)\} \\
\mathcal{O}.\mathbf{I} & = & \{\mathsf{reccom}(cId, recommendation)\} \\
\mathcal{O}.\mathbf{S} & = & \{application(cId, loan), \\
 & & awaitsHist(cId, ssn, name, loan, rating), \\
 & & awaitsMgr(cId, ssn, name, loan, rating, \\
 & & \qquad account, balance)\} \\
\mathcal{O}.\mathbf{A} & = & \{\ \mathrm{letter}(cId, name, loan, decision)\} \\
\mathcal{O}.\mathbf{Q^f_{in}} & = & \{\mathbf{apply}(cId, loan), \mathbf{decision}(cId, dec), \\
 & & \mathbf{rating}(ssn, category)\} \\
\mathcal{O}.\mathbf{Q^n_{in}} & = & \{\mathbf{history}(ssn, account, balance)\} \\
\mathcal{O}.\mathbf{Q^f_{out}} & = & \{\mathbf{getRating}(ssn), \mathbf{getHistory}(ssn)\} \\
\mathcal{O}.\mathbf{Q^n_{out}} & = & \{\mathbf{recommend}(cId, loan, decision, rating, \\
 & & \qquad account, balance)\}
\end{array}$$

We show some of $\mathcal{O}$'s rules below. $\mathcal{O}$ runs on top of a customer database which records each customer's id, ssn and name. The input reccom allows the officer to provide an approval or denial recommendation for any customer, by picking from a menu generated by input rule (1). Upon arrival of an application message, $\mathcal{O}$ reacts automatically, without the officer's involvement, as follows. The application message is saved in the *application* state (rule (2)) and a credit rating request message is sent to the credit agency peer $\mathcal{CR}$ (rule (3)). Notice how the customer database is consulted to translate the bank-specific customer id to the ssn required by the credit agency's Web service. On receipt of a message rating a customer's credit as "excellent", an approval letter is generated (4). Customers with "poor" rating get denial letters (5). For all other ratings, a message is sent to request the credit history details, namely the list of open accounts and their balance (rule (7)). In addition, the customer's information and rating are recorded in the state *awaitsHist* (rule (8)), where they await the response of $\mathcal{CR}$. Upon its receipt, the complete customer information gathered so far is recorded in the state *awaitsMgr* (rule (9)). The subsequent input of a recommendation by the officer triggers the sending of a recommendation message to the manager's Web service $\mathcal{M}$ (rule (10)). $\mathcal{M}$'s reply causes an appropriate letter-writing action (6).

$$\text{Options}_{\mathsf{reccom}}(id,rec) \leftarrow \exists ssn,name\ \underline{\mathsf{customer}}(id,ssn,name) \wedge (rec = \text{“}approve\text{”} \vee rec = \text{“}deny\text{”}) \quad (1)$$

$$application(id,loan) \leftarrow ?\mathbf{apply}(id,loan) \quad (2)$$

$$!\mathbf{getRating}(ssn) \leftarrow \exists id,loan,name\ ?\mathbf{apply}(id,loan) \wedge \underline{\mathsf{customer}}(id,ssn,name) \quad (3)$$

$$\begin{aligned}\text{letter}(id,name,loan,dec) \leftarrow & \exists ssn\ \underline{\mathsf{customer}}(id,ssn,name) \\ & \wedge application(id,loan) \\ & \wedge[?\mathbf{rating}(ssn, \text{“}excellent\text{”}) \wedge dec = \text{“}approved\text{”} & (4)\\ & \vee ?\mathbf{rating}(ssn, \text{“}poor\text{”}) \wedge dec = \text{“}denied\text{”} & (5)\\ & \vee ?\mathbf{decision}(id,dec)] & (6)\end{aligned}$$

$$!\mathbf{getHistory}(ssn) \leftarrow \exists r\ ?\mathbf{rating}(ssn,r) \wedge \neg(r = \text{“}excellent\text{”} \vee r = \text{“}poor\text{”}) \quad (7)$$

$$\begin{aligned}awaitsHist(id,ssn,name,l,r) \leftarrow & ?\mathbf{rating}(ssn,r) \wedge \\ & \neg(r = \text{“}excellent\text{”} \vee r = \text{“}poor\text{”}) \wedge application(id,l) \\ & \wedge \underline{\mathsf{customer}}(id,ssn,name) & (8)\end{aligned}$$

$$\begin{aligned}awaitsMgr(id,ssn,name,loan,rating,acc,bal) \leftarrow & \\ & ?\mathbf{history}(ssn,acc,bal) \\ & \wedge\ awaitsHist(id,ssn,name,loan,rating) & (9)\end{aligned}$$

$$\begin{aligned}!\mathbf{recommend}(id,ssn,name,loan,rec,rating,acc,bal) \leftarrow & \\ & \mathsf{reccom}(id,rec) \\ & \wedge awaitsMgr(id,loan,ssn,name,rating,acc,bal) & (10)\end{aligned}$$

□

We next define the notion of a configuration of a Web service, and the transition relation among configurations. Informally, a configuration consists of the fixed database, the contents of the message queues, as well as the states, previous and current inputs, and actions. To formalize the usage of message queues, we first introduce the following notation, allowing us to refer to the first and last message in each in-queue. Suppose Q^{in} is an instance of the in-queues, i.e. a mapping associating to each $R \in \mathbf{Q_{in}}$ a finite sequence of instances of R. Given Q^{in}, we define the relational instances $f(Q^{in})$ and $l(Q^{in})$, both of schema $\mathbf{Q_{in}}$, holding the first, respectively last messages from all queues. Thus, if $R \in \mathbf{Q^f_{in}}$ ($Q^{in}(R)$ is a flat queue), then $f(Q^{in})(R)$ and $l(Q^{in})(R)$ contain each a singleton tuple which is the first, resp. last message in $Q^{in}(R)$. If $R \in \mathbf{Q^n_{in}}$, $f(Q^{in})(R)$ and $l(Q^{in})(R)$ contain the set of tuples making up the first, resp. last message of $Q^{in}(R)$. If $Q^{in}(R)$ is empty so are $f(Q^{in})(R)$ and $l(Q^{in})(R)$.

Definition 2.3. *Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{Q_{in}}, \mathbf{Q_{out}}, \mathcal{R}\rangle$ be a Web service. A* configuration *of $\mathcal{W}$ is a tuple $\langle D, S, I, P, A, Q^{in}, Q^{out}\rangle$ where the database D is an instance of $\mathbf{D}$, the state S is an instance of $\mathbf{S}$, the input I is an instance of $\mathbf{I}$, the previous input P is an instance of $\mathbf{Prev_I}$, and the action A is an instance of $\mathbf{A}$. Additionally, Q^{in} (Q^{out}) associates to each $R \in \mathbf{Q_{in}}$ ($R \in \mathbf{Q_{out}}$) a finite sequence $Q^{in}(R)$ ($Q^{out}(R)$) of instances of R. We refer to Q^{in} and Q^{out} respectively as instances of the in-queues and out-queues. For each $R \in \mathbf{Q_{in}}$, the queue state $empty_R$ is true iff the sequence associated to R by Q^{in} is empty. Finally, for each relation R in $\mathbf{I}$ of arity $k > 0$, $I(R) \subseteq \{v\}$ for some $v \in Options_R$, where $Options_R$ is the result of evaluating φ_R on D, S, $f(Q^{in})$, and P; if R has arity zero (so R is a propositional state), then $I(R)$ is a truth value such that $I(R) \rightarrow Options_R$.*

We next define the transition relation of a Web service. This relation defines for every current configuration of the Web service its legal successor configurations, reachable in one atomic step.

Definition 2.4. *Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{Q_{in}}, \mathbf{Q_{out}}, \mathcal{R}\rangle$ be a Web service and $C_i = \langle D_i, S_i, I_i, P_i, A_i, Q^{in}_i, Q^{out}_i\rangle$, a configuration of $\mathcal{W}$. A configuration*

$$C_{i+1} = \langle D_{i+1}, S_{i+1}, I_{i+1}, P_{i+1}, A_{i+1}, Q^{in}_{i+1}, Q^{out}_{i+1}\rangle$$

of $\mathcal{W}$ is a legal successor *configuration of C_i iff:*

- *$D_i = D_{i+1}$ i.e. the database remains unchanged. We will denote it with D.*
- *for each relation $prev_R$ in $\mathbf{Prev_I}$, $P_{i+1}(prev_R) = I_i(R)$ if $I_i(R) \neq \emptyset$, and $P_{i+1}(prev_R) = P_i(prev_R)$ otherwise.*
- *for each relation S in $\mathbf{S}$ that is not a queue state, $S_{i+1}(S)$ is the result of evaluating*

  $$\begin{array}{l}(\varphi^+_S(\bar{x}) \wedge \neg\varphi^-_S(\bar{x})) \vee \\ (S(\bar{x}) \wedge \varphi^-_S(\bar{x}) \wedge \varphi^+_S(\bar{x})) \vee \\ (S(\bar{x}) \wedge \neg\varphi^-_S(\bar{x}) \wedge \neg\varphi^+_S(\bar{x}))\end{array}$$

  on D, S_i, $f(Q^{in}_i)$, I_i, and P_i, where $\varphi^\epsilon_S(\bar{x})$ is taken to be false *if it is not provided ($\epsilon \in \{+,-\}$). In particular, S remains unchanged if no insertion or deletion rule is specified for it.*
- *for each relation A in $\mathbf{A}$, $A_{i+1}(A)$ is the result of evaluating φ_A on D, S_i, $f(Q^{in}_i)$, I_i, and P_i.*
- *for each relation R in $\mathbf{Q_{out}}$, let m_R denote the result of evaluating φ_R on D, S_i, $f(Q^{in}_i)$, I_i, and P_i. If $R \in \mathbf{Q^n_{out}}$, $Q^{out}_{i+1}(R)$ is obtained by enqueuing m_R into $Q^{out}_i(R)$. If $R \in \mathbf{Q^f_{out}}$, then if m_R is non-empty, $Q^{out}_{i+1}(R)$ is obtained by enqueuing into $Q^{out}_i(R)$ a singleton containing a non-deterministically picked tuple from m_R. If m_R is empty, $Q^{out}_{i+1}(R) = Q^{out}_i(R)$ so the queue remains unchanged.*
- *for each relation R in $\mathbf{Q_{in}}$, if R is mentioned in the set of rules $\mathcal{R}$, the corresponding new in-queue $Q^{in}_{i+1}(R)$ is obtained by dequeuing the first message from $Q^{in}_i(R)$. Otherwise, $Q^{in}_{i+1}(R) = Q^{in}_i(R)$.*

Notice that we employ a “snapshot” semantics, in the sense that at each step all rules are simultaneously interpreted over the current configuration. In particular, all occurrences of $R \in \mathbf{Q_{in}}$ in $\mathcal{R}$ refer to the same first message in q_R. Also notice that a message sent in the current configuration i is enqueued at the receiver in the successor configuration $i+1$. Furthermore, since $\mathcal{W}$'s out-queues are some other peers' in-queues, $\mathcal{W}$ side-effects the receiver's queues upon transitioning to the successor configuration.

We next define the syntax and semantics of a *composition* of Web services.

Definition 2.5. *A* composition *is a set of Web services $\mathcal{C} = \{\mathcal{W}_i\}_{1\le i\le n}$ such that $\mathcal{W}_i.\mathbf{Q_{in}} \cap \mathcal{W}_j.\mathbf{Q_{in}} = \emptyset$ and $\mathcal{W}_i.\mathbf{Q_{out}} \cap \mathcal{W}_j.\mathbf{Q_{out}} = \emptyset$ for $i \neq j$, so each queue is an output (input) queue for at most one peer. We say that $\mathcal{C}$ is* closed *if $\bigcup_{i=1}^n \mathcal{W}_i.\mathbf{Q_{in}} = \bigcup_{i=1}^n \mathcal{W}_i.\mathbf{Q_{out}}$, otherwise it is* open. *We refer to the composition members as* peers.

Intuitively, a composition is closed if every peer's in-queue is some other peer's out-queue and conversely, so that no input messages are received from or sent outside the composition.

Since we are interested in verifying the correctness of a composition's behavior, we describe the latter via the notion of "run" of a closed composition. Essentially, a run is a sequence of snapshots through which the composition evolves. Each snapshot specifies the current configuration of each peer. We consider only serialized runs, i.e. runs in which at every step precisely one peer performs a transition.

Definition 2.6. *Let $\mathcal{C}$ be a closed composition $\{\mathcal{W}_j\}_{1\le j\le n}$ and $\overline{D} = \{D_j\}_{1\le j\le n}$ be database instances where each D_j is an instance of $\mathcal{W}_j.\mathbf{D}$. A run of $\mathcal{C}$ over $\overline{D}$ is an infinite sequence $\{(k_i, \langle C_i^j\rangle_{j\in[1,n]})\}_{i\ge 0}$, where $k_i \in [1,n]$ and C_i^j is a configuration of peer $\mathcal{W}_j$, such that:*

- *for each $j \in [1,n]$, the state, action, previous input and queues of C_0^j are empty and its database is D_j;*
- *for each $i \ge 0$, $j, l \in [1,n]$ and for each $R \in \mathcal{W}_j.\mathbf{Q_{out}} \cap \mathcal{W}_l.\mathbf{Q_{in}}$, the queues associated to R in C_i^j and C_i^l are identical;*
- *for each $i \ge 0$,*
  - *(i) $C_{i+1}^{k_i}$ is a legal successor configuration of $C_i^{k_i}$ for peer $\mathcal{W}_{k_i}$, and*
  - *(ii) for each $1 \le l \le n$ with $l \neq k_i$, the database, input, previous input, state and actions of C_{i+1}^l and C_i^l coincide, and so do the queues for each $R \in (\mathcal{W}_l.\mathbf{Q_{in}} \setminus \mathcal{W}_{k_i}.\mathbf{Q_{out}} \cup \mathcal{W}_l.\mathbf{Q_{out}} \setminus \mathcal{W}_{k_i}.\mathbf{Q_{in}})$.*

We say that (only) peer $\mathcal{W}_{k_i}$ moves *at step i. $\langle C_i^j\rangle_{1\le j\le n}$ is called the* snapshot *at step i.*

Intuitively, at step i in the run, only the peer $\mathcal{W}_{k_i}$ moves while the others wait their turn. While waiting, their configurations are preserved, except for the queues which are updated by peer $\mathcal{W}_{k_i}$. These are the queues into which $\mathcal{W}_{k_i}$ sends, and from which $\mathcal{W}_{k_i}$ receives.

Lossy and perfect channels The semantics of Web compositions as defined above assumes that channels are *perfect*, i.e. all messages sent across a channel are received. This is modeled by enqueuing each sent message into the corresponding in-queue. In practice however, channels are often *lossy*, i.e. messages may be lost in transit. Indeed, this is reflected in the models used in standard work on communicating finite-state automata [2, 3]. We can also define a variant of the semantics for Web compositions that captures lossy channels, by non-deterministically allowing sent messages to not be enqueued in the corresponding in-queue. We refer to such Web compositions as *Web compositions with lossy channels*, and to the Web compositions defined above as *Web compositions with perfect channels*. As we shall see, many of the results depend on whether channels are lossy or perfect.

There are many ways in which the correctness of a composition can be specified. We investigate two alternatives next: temporal logic (Section 3) and conversation protocols (Section 4).

3. LTL-FO PROPERTIES

The correctness of a composition is specified by statements which can express the properties of individual run snapshots (using First-Order Logic), as well as the temporal relationship among these snapshots (using Linear Temporal Logic operators). We call these statements LTL-FO properties.

Definition 3.1. *(Inspired by [14, 4, 27]) The language LTL-FO (first-order linear-time temporal logic) is obtained by closing FO under negation, disjunction, and the following formula formation rule: If φ and ψ are formulas, then $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\psi$ are formulas. Free and bound variables are defined in the obvious way. The* universal closure *of an LTL-FO formula $\varphi(\bar{x})$ with free variables $\bar{x}$ is the formula $\forall\bar{x}\varphi(\bar{x})$. An LTL-FO sentence is the universal closure of an LTL-FO formula.*

Note that quantifiers cannot be applied to formulas containing temporal operators, except by taking the universal closure of the entire formula, yielding an LTL-FO sentence.

Composition Schema. Let $\mathcal{C} = \{\mathcal{W}_j\}_{1\le j\le n}$ be a composition. Properties of runs are expressed over the *composition schema* of $\mathcal{C}$ which consists of

- the union of all peer schemas in which each relation is qualified by the name of its peer: $\mathcal{C}.\mathbf{Y} = \bigcup_{j=1}^n \{\mathcal{W}_j.R \mid R \in \mathcal{W}_j.\mathbf{Y}\}$ for each $\mathbf{Y} \in \{\mathbf{I}, \mathbf{prev_I}, \mathbf{S}, \mathbf{A}, \mathbf{Q_{in}}, \mathbf{Q_{in}^f}, \mathbf{Q_{in}^n}, \mathbf{Q_{out}}, \mathbf{Q_{out}^f}, \mathbf{Q_{out}^n}\}$;
- the set of propositional states $\{move_{\mathcal{W}} \mid \mathcal{W} \in \mathcal{C}\}$. Intuitively, at every step of a run, $move_{\mathcal{W}}$ holds iff $\mathcal{W}$ is the moving peer at that step.

Semantics of LTL-FO Properties. Let $\psi = \forall\bar{x}\varphi(\bar{x})$ be an LTL-FO sentence over the above schema. We say that the composition $\mathcal{C}$ satisfies $\forall\bar{x}\varphi(\bar{x})$ (denoted $\mathcal{C} \models \psi$) iff every run ρ of $\mathcal{C}$ satisfies ψ. Let $\rho = \{\rho_i\}_{i\ge 0}$ be a run of $\mathcal{C}$ over databases $\bar{D}$, and let $\rho_{\ge j}$ denote $\{\rho_i\}_{i\ge j}$, for $j \ge 0$. Note that $\rho = \rho_{\ge 0}$. Let Dom(ρ) be the active domain of ρ, i.e. the set of all elements occurring in relations or as constants in ρ. The run ρ satisfies $\forall\bar{x}\varphi(\bar{x})$ (denoted $\rho \models \forall\bar{x}\varphi(\bar{x})$) iff for each valuation ν of $\bar{x}$ in Dom(ρ), $\rho_{\ge 0}$ satisfies $\varphi(\nu(\bar{x}))$. The latter is defined by structural induction on the formula: An FO sentence ψ is satisfied by $\rho_i = (k_i, \langle C_i^j\rangle_{1\le j\le n})$ if the structure ρ_i' satisfies ψ, where ρ_i' is obtained from ρ_i by

- replacing the instance q of in-queues in ρ_i with the relational instance $f(q)$;
- replacing the instance q of out-queues of ρ_i with the relational instance $l(q)$;
- setting the propositional state $move_{\mathcal{W}_{k_i}}$ to *true* and $move_{\mathcal{W}}$ for $\mathcal{W} \neq \mathcal{W}_{k_i}$ to *false*.

Intuitively, an in-queue symbol Q used in an LTL-FO formula is taken to refer to the first message of Q, currently available as an input message, and an out-queue symbol Q refers to the message most recently added to the queue Q. Note that, in order to refer to the output messages generated at step i, one has to refer to the last messages of the out-queues at step $i+1$.

The semantics of Boolean operators is the obvious one. The meaning of the temporal operators **X, U** is the following (where $\models$ denotes satisfaction and $j \geq 0$):

- $\rho_{\geq j} \models \mathbf{X}\varphi$ iff $\rho_{\geq j+1} \models \varphi$,
- $\rho_{\geq j} \models \varphi\mathbf{U}\psi$ iff $\exists k \geq j$ such that $\rho_{\geq k} \models \psi$ and $\rho_{\geq l} \models \varphi$ for $j \leq l < k$.

Observe that the above temporal operators can simulate all commonly used operators, including **B** (before), **G** (always) and **F** (eventually). Indeed, $\varphi\mathbf{B}\psi$ ("φ must hold *before* ψ fails") is equivalent to $\neg(\neg\varphi\mathbf{U}\neg\psi)$; $\mathbf{G}\varphi$ ("φ *generally* holds") is equivalent to *false* $\mathbf{B}\ \varphi$; $\mathbf{F}\varphi$ ("φ *finally* holds") is expressible as *true* $\mathbf{U}\ \varphi$. We use the above operators as shorthand in LTL-FO formulas whenever convenient.

Example **3.2** LTL-FO sentences can express many interesting properties of compositions. For instance, property (11) below states that every received application message from an applicant found in the customer database will eventually result in either an approval or a denial letter.

$$\begin{gathered}\forall id,l,name,ssn\ \mathbf{G}\\ [(\mathcal{O}.?\mathbf{apply}(id,l) \wedge \mathcal{O}.\underline{\mathsf{customer}}(id,ssn,name))\\ \rightarrow \mathbf{F}\\ (\mathcal{O}.\ \text{letter}(id,name,l,\text{“denied”})\\ \vee\mathcal{O}.\ \text{letter}(id,name,l,\text{“approved”}))]\end{gathered} \qquad (11)$$

The following requires loans to be approved only for applicants with excellent credit rating or for those previously cleared by the manager.

$$\begin{gathered}\forall id,name,loan\ \mathbf{G}\\ [(\exists ssn\ \mathcal{CR}.!\mathbf{rating}(ssn,\text{“excellent”}) \wedge \mathcal{O}.\underline{\mathsf{customer}}(id,ssn,name)\\ \vee\ \mathcal{M}.!\mathbf{decision}(id,\text{“approved”}))\\ \mathbf{B}\\ \neg\mathcal{O}.\ \text{letter}(id,name,loan,\text{“approved”})]\end{gathered}$$

□

3.1 Decidable Verification

In this section we establish two restrictions under which verification is decidable for a significant class of compositions with lossy channels. These restrictions will be justified in Section 3.2, where we show that even modest relaxations lead to undecidability. The first restriction is syntactic and is called *input-boundedness*. The second is of a semantic nature, assuming *bounded-length queues*.

Input-boundedness is a natural restriction inspired by the observation that each peer is driven by the user input and by the incoming messages. Essentially, we require that quantified variables range only over the active domain of the current inputs, the previous inputs and the first messages of flat (but not nested!) queues. This restriction is enforced syntactically as follows.

The set of *input-bounded* FO formulas over a composition $\mathcal{C}$'s schema is obtained by replacing in the definition of FO the quantification formation rule with the following:

- if φ is an input-bounded formula, α is an atom using a relational symbol from $\mathcal{C}.\mathbf{I} \cup \mathcal{C}.\mathbf{Prev_I} \cup \mathcal{C}.\mathbf{Q^f_{in}} \cup \mathcal{C}.\mathbf{Q^f_{out}}$, $\bar{x} \subseteq free(\alpha)$, and $\bar{x} \cap free(\beta) = \emptyset$ for every state, action or nested in-queue atom β in φ, then $\exists\bar{x}(\alpha \wedge \varphi)$ and $\forall\bar{x}(\alpha \rightarrow \varphi)$ are input-bounded formulas.

A peer is input-bounded iff

1. all state, action, and send rules into nested queues are given by input-bounded formulas, and
2. all input rules, as well as all send rules into flat queues use $\exists^*$FO formulas in which all state and nested queue atoms are ground.

An LTL-FO sentence over the composition schema is input-bounded iff all of its FO subformulas are input-bounded.

Example **3.3** Peer $\mathcal{O}$ in Example 2.2 is input-bounded, and so are the properties in Example 3.2. Examples for non-input-bounded properties can be found in [13]. □

For the particular case when the composition consists of a single peer without any message queues, the above restriction degenerates to the notion of input-bounded Web service from [13]. To show that input-bounded specifications of individual peers cover a large class of applications, we have modeled significant parts of a computer shopping Web site similar to the Dell computer shopping site, an airline reservation site similar to Expedia, an online bookstore in the spirit of Barnes & Noble, and a sports Web site on the Motorcycle Grand Prix (all published at [1]).

As it turns out, for proper compositions the syntactic input-boundedness restriction is insufficient to yield decidability. We need to make the further assumption that the queues are bounded. We say that a composition has k-*bounded queues* if each queue may simultaneously contain at most k messages. Messages arriving when the receiver's in-queue is full are simply dropped. With these restrictions, we can state our main decidability result:

Theorem 3.4. *It is decidable whether an input-bounded composition with k-bounded queues and lossy channels satisfies an input-bounded LTL-FO property. Furthermore, the problem is* PSPACE*-complete for schemas with fixed bound on the arity, and* EXPSPACE *otherwise.*

The proof is outlined in Appendix A. It essentially consists in a non-trivial PTIME-reduction to the problem of verifying input-bounded properties of compositions consisting of a single peer with no queues. In addition, the peer can inspect, for each input I, the k previous non-empty inputs to I, using relations $prev^i_I$ for $1 \leq i \leq k$ (instead of just the immediately previous non-empty input $prev_I$, as in our definition of a peer). We refer to peers with this ability as peers with k-lookback. The decidability of verification for peers with k-lookback is shown by adapting and extending Theorem 3.5 in [13], as follows.

Lemma 3.5. **(follows from [13])** *It is decidable, given a composition $\mathcal{C} = \{\mathcal{W}\}$ where $\mathcal{W}$ is an input-bounded peer with k-lookback and no message queues and an input-bounded LTL-FO property φ, whether $\mathcal{C}$ satisfies φ. The problem is* PSPACE*-complete for schemas with fixed bound on the arity, and* EXPSPACE *otherwise.*

Remarks *Transmission delays* In our model of compositions, we assumed instantaneous transmission of messages (that are not lost) by having each message be enqueued immediately after being sent. The model and results can be easily adapted to model arbitrary transmission delays, as long as the capacity of each channel is bounded. In the simulation of the composition by a single peer, the transmission delays can be captured by partitioning each in-queue into a received portion followed by an in-transit portion, and triggering the transition of a message from the in-transit portion to the received portion using a new propositional input turned on non-deterministically by a dummy user. The decidability does not extend to channels with unbounded capacity. Indeed, this is similar to having unbounded queues, for which verification is undecidable (see Corollary 3.6).
Perfect nested message channels Theorem 3.4 assumes that all channels are lossy. It turns out that the result still holds if nested message channels are perfect, and flat message channels are lossy. The proof (see Appendix) is the same, except that the input lost_Q used to simulate the loss of a nested message is removed.

3.2 Boundaries of Verification Decidability

We have shown in Section 3.1 that we can soundly and completely verify a significant and expressive class of compositions and properties. It is natural to ask whether the restrictions of Theorem 3.4 are truly necessary. In this section we show that this is indeed the case, in the sense that minimally relaxing any single restriction leads to undecidability. The proofs are all in Appendix A.

We first investigate the assumptions pertaining to the boundedness of queues and lossyness of channels. We immediately obtain the following as consequences of prior work on peers which are communicating finite-state machines (CFSM) with queues holding propositional messages: verification is undecidable for unbounded queues, whether they are perfect (Brand and Zafiropulo [6]) or lossy (Abdulla and Jonson [2]):

Corollary 3.6. **(of [6, 2])** *It is undecidable to determine if an input-bounded property is satisfied by an input-bounded composition $\mathcal{C}$ with unbounded queues, regardless of whether they are lossy or perfect.*

According to Corollary 3.6, the unbounded-queue assumption alone suffices to cause undecidability. Theorem 3.7 below provides the complementary result, showing that the perfect-queue assumption is sufficient for undecidability even when all queues are bounded. This result highlights the impact of data-awareness on the verification problem. Contrast it with the finite-state case, in which the composition of CFSMs via bounded, perfect queues is easily reducible to a single FSM, for which verification is decidable.

Theorem 3.7. *It is undecidable whether an input-bounded property is satisfied by an input-bounded composition with no nested queues and 1-bounded,* perfect *flat queues.*

Recall that the semantics of sending into flat queues requires a message to be non-deterministically picked whenever the send rule generates several candidates. We consider a plausible alternative semantics in which the generation of multiple candidate messages is treated as a run-time error, in the sense that no message is sent and an error flag is set instead. To this end, we extend the schema of each peer $\mathcal{W}$ with a propositional state $error_R$ for each flat out-queue $R \in \mathcal{W}.\mathbf{Q^f_{out}}$. This state is appropriately set by the legal successor relation and it can be consulted by the peer rules and the properties. We say that the flat queues in this peer flavor have *deterministic send* rules.

Theorem 3.8. *It is undecidable whether an input-bounded property is satisfied by an input-bounded composition of peers with no nested queues, and 1-bounded lossy flat queues with* deterministic send *rules.*

We next consider a list of minor relaxations of the syntactic input-boundedness restriction (Section 3.1). First, we focus on the restriction disallowing quantified variables to appear in nested queue atoms. Notice that a consequence of this restriction is the impossibility to test emptiness of messages received via nested queues: if $R \in \mathcal{W}.\mathbf{Q^n_{in}}$, $\exists \bar{x}\ ?\mathbf{R}(\bar{x})$ checks the non-emptiness of the message received along queue q_R. Also notice that this test is legal if $R \in \mathbf{Q^f_{in}}$. Consider a relaxation of input-boundedness allowing access to a built-in predicate *empty*$(?\mathbf{R})$ that is set to true iff the first message in q_R is non-empty. We show that this is enough to yield undecidability, even if the emptiness tests are only used in formulating the property to be verified.

Theorem 3.9. *It is undecidable whether an input-bounded property* with emptiness tests on nested messages *is satisfied by an input-bounded composition $\mathcal{C}$ with 1-bounded queues (lossy or perfect).*

A second restriction imposed by input-boundedness requires the nested queue atoms appearing in input and in flat queue send rules to be *ground* (contain terms constructed only of constants, no variables). Its removal leads to undecidability.

Theorem 3.10. *It is undecidable whether an input-bounded property is satisfied by a composition $\mathcal{C}$ with 1-bounded queues and lossy flat queues, where $\mathcal{C}$ is input-bounded except for allowing* non-ground nested in-queue atoms *in input rules, or in flat queue send rules.*

The proof of Theorem 3.10 follows from an easy modification of the proof of a result in [13] and is omitted.

4. CONVERSATION PROTOCOLS

The most prominent kind of correctness property for compositions considered in previous work is the notion of *(finite-state) conversation protocol.* It requires that the sequence of messages as observed by some global observer belongs to an ω-regular language accepted by a Büchi automaton. The notion was introduced by Fu, Bultan and Su [17], as a generalization of an earlier version proposed by an industrial standard (IBM's conversation support project [19]) for the model of communicating finite-state machines (CFSMs) sending and receiving propositional messages via queues.

Conversation protocols of the above flavor can also be verified for our compositions. We refer to such protocols as "data-agnostic", because they ignore the contents of messages, checking only the sequence of observed message *names.* For instance, in our running example, a data-agnostic protocol would require any **getHistory** message to be followed by a **history** message. Alternatively, we may consider a "data-aware" extension of conversation protocols, where the contents of messages is taken into account in the specification of the protocol. We first show how data-agnostic protocols can be verified for compositions, then consider data-aware protocols.

In the context of lossy channels and bounded queues, several semantics for conversation protocols are possible. One is to ignore dropped messages and only consider messages actually being enqueued. Intuitively, this places the observer of messages at the recipients. We refer to this semantics as *observer-at-recipient*. An alternative would be to observe all sent messages, regardless of whether or not they are enqueued. This corresponds to placing the observer at the source of sent messages. We refer to this semantics as *observer-at-source*. We first consider data-agnostic conversation protocols with observer-at-recipient semantics.

Let $\mathcal{C}$ be a composition and $\Sigma = \mathcal{C}.\mathbf{Q_{out}}$. A data-agnostic conversation protocol for $\mathcal{C}$ is a pair $(\Sigma, \mathcal{B})$ where $\mathcal{B}$ is a Büchi automaton over alphabet Σ.[2] Satisfaction of $(\Sigma, \mathcal{B})$ by $\mathcal{C}$ under the observer-at-recipient semantics is defined as follows. Let $\rho = \{\rho_i\}_{i \geq 0}$ be a run of $\mathcal{C}$. We first define the set of propositions Q in Σ satisfied by each configuration of the run. The configuration ρ_0 satisfies no proposition Q. A configuration ρ_i, $i > 0$, satisfies Q if a new message is placed in the queue for Q in the transition from ρ_{i-1} to ρ_i. Let $\sigma(\rho_i)$ provide the subset of Σ satisfied by ρ_i, $i \geq 0$. The run ρ satisfies the protocol iff $\{\sigma(\rho_i)\}_{i \geq 0}$ is accepted by $\mathcal{B}$. The composition $\mathcal{C}$ satisfies the protocol $(\Sigma, \mathcal{B})$ iff every run of $\mathcal{C}$ satisfies the protocol. Clearly, data-agnostic conversation protocols for our infinite-state compositions strictly generalize conversation protocols for finite-state systems.

EXAMPLE **4.1** Since Büchi automata are strictly more expressive than LTL [28, 9], conversation protocols include properties expressible in LTL. For presentation simplicity, we illustrate an LTL-expressible protocol which, in our running example, requires each credit rating request message **getRating** to be followed by a **rating** reply:

$$\mathbf{G}(\mathbf{getRating} \rightarrow \mathbf{F}\ \mathbf{rating}).$$

□

THEOREM 4.2. *It is decidable whether an input-bounded composition with bounded queues and lossy channels satisfies a data-agnostic conversation protocol with observer-at-recipient semantics. The problem is* PSPACE*-complete for schemas with bounded arity, and* EXPSPACE *otherwise.*

Note that PSPACE decidability is reasonable as far as verification goes, given that LTL verification of finite-state (Mealy) machines is already PSPACE-complete [9].

Suppose we wish to verify protocols under the alternative observer-at-source semantics. Unfortunately, verification is undecidable in this case.

THEOREM 4.3. *It is undecidable, given an input-bounded composition $\mathcal{C}$ with bounded queues and lossy channels, and a data-agnostic conversation protocol $(\Sigma, \mathcal{B})$ with observer-at-source semantics, whether $\mathcal{C}$ satisfies $(\Sigma, \mathcal{B})$.*

We next consider data-aware conversation protocols. In view of Theorem 4.3, we only consider protocols with observer-at-recipient semantics, which is assumed by default. The data-aware protocols generalize finite-state protocols, the data-agnostic protocols discussed above, and also input-bounded LTL-FO properties over the schema $\mathcal{C}.\mathbf{Q_{out}}$. We show the decidability of checking compliance of a composition with respect to the resulting protocol.

DEFINITION 4.4. *A* data-aware conversation protocol *over the schema of composition $\mathcal{C}$ is a triple $(\Sigma, \mathcal{B}, \{\varphi_\sigma\}_{\sigma\in\Sigma})$, where: Σ is a set of propositional symbols, $\{\varphi_\sigma\}_{\sigma\in\Sigma}$ is a family of FO formulas over schema $\mathcal{C}.\mathbf{Q_{out}}$, one for each symbol in Σ, and $\mathcal{B}$ is a Büchi automaton with transitions guarded by boolean formulas over Σ.*

Intuitively, we use the symbols in $\sigma \in \Sigma$ as shorthands for formulas φ_σ over the current snapshot.

We next define the semantics of data-aware conversation protocols of the observer-at-recipient flavor. Recall that the semantics of FO formulas interprets each $Q \in \mathcal{C}.\mathbf{Q_{out}}$ as the message last placed in the queue for Q. This is consistent with the observer-at-recipient semantics for protocols.

Let $\mathcal{P}$ be a protocol $(\Sigma, \mathcal{B}, \{\varphi_\sigma\}_{\sigma\in\Sigma})$. If each φ_σ is a sentence, a run ρ of composition $\mathcal{C}$ satisfies $\mathcal{P}$ iff $\mathcal{B}$ accepts the infinite sequence obtained by computing the truth values for $\{\varphi_\sigma\}_{\sigma\in\Sigma}$ in each snapshot of ρ. If the φ_σ's are formulas with free variables, let $\bar{x} := \bigcup_{\sigma\in\Sigma} \mathit{freeVars}(\varphi_\sigma)$. Denoting with $\mathbf{Dom}(\rho)$ the active domain of ρ, ρ satisfies $\mathcal{P}$ iff for each valuation ν of $\bar{x}$ in $\mathbf{Dom}(\rho)$, ρ satisfies the protocol (without free variables) $(\Sigma, \mathcal{B}, \{\varphi_\sigma(\nu(\bar{x}))\}_{\sigma\in\Sigma})$. We say that composition $\mathcal{C}$ satisfies conversation protocol $\mathcal{P}$ iff every run ρ of $\mathcal{C}$ satisfies $\mathcal{P}$.

Note that any LTL-FO property over schema $\mathcal{C}.\mathbf{Q_{out}}$ can be expressed by a data-aware conversation protocol, while the converse is not true. This observation follows from well-known results relating the expressivity of propositional LTL and Büchi automata [28]. An example data-aware conversation protocol is property (12) in Example 5.1 below.

We say that conversation protocol $\mathcal{P} = (\Sigma, \mathcal{B}, \{\varphi_\sigma\}_{\sigma\in\Sigma})$ is *input-bounded* if each φ_σ is input bounded.

THEOREM 4.5. *It is decidable whether an input-bounded composition with bounded queues and lossy channels satisfies an input-bounded data-aware conversation protocol. The problem is* PSPACE*-complete for schemas with bounded arity, and* EXPSPACE *otherwise.*

Boundary of decidability. As in the case of LTL-FO properties over the entire schema, even small relaxations of the restrictions under which the above decidability results were obtained lead to undecidability of verification. We consider relaxations similar to those in Section 3.2.

THEOREM 4.6. *The following are undecidable:*

(i) satisfaction of a data-agnostic protocol by an input-bounded composition with unbounded queues.

(ii) satisfaction of a data-agnostic conversation protocol by an input-bounded composition with no nested queues and 1-bounded, perfect *flat queues.*

(iii) satisfaction of a data-aware conversation protocol with ground message parameters, by an input-bounded composition with deterministic *lossy flat 1-bounded queues and perfect nested 1-bounded queues.*

(iv) satisfaction of a data-agnostic conversation protocol augmented with emptiness tests on nested messages, by an input-bounded composition with (perfect or lossy) 1-bounded queues.

[2]A Büchi automaton is a finite-state automaton accepting infinite sequences iff they drive the automaton to visit some final state infinitely often [28].

5. MODULAR VERIFICATION

It is often useful to verify a composition $\mathcal{C}$ in a modular fashion, i.e. to verify that a subset of its peers behaves correctly when the full specification of the other peers is not available and the only knowledge about them is declared in the form of properties of their input-output behavior. Such verification is the best one can hope for when the various peers are provided by autonomous parties unwilling to disclose the internal implementation details. Even when all peers are owned by a single party, modular verification enables the validation of a peer subset before the design of the others is completed.

Recall that a set of peers $\mathcal{C}$ is an open composition if $\mathcal{C}.\mathbf{Q_{in}} \neq \mathcal{C}.\mathbf{Q_{out}}$. In particular, any single peer with at least one queue is an open composition. Notice that $\mathcal{C}$ interacts with outside peers by means of the message queues in the symmetric difference $\mathcal{C}.\mathbf{Q_{in}} \,\Delta\, \mathcal{C}.\mathbf{Q_{out}} = (\mathcal{C}.\mathbf{Q_{in}} \setminus \mathcal{C}.\mathbf{Q_{out}}) \cup (\mathcal{C}.\mathbf{Q_{out}} \setminus \mathcal{C}.\mathbf{Q_{in}})$. The queues in $\mathcal{C}.\mathbf{Q_{in}} \setminus \mathcal{C}.\mathbf{Q_{out}}$ hold messages output by the environment, and are denoted $\mathcal{E}.\mathbf{Q_{out}}$, where we denote the environment with $\mathcal{E}$. The queues in $\mathcal{C}.\mathbf{Q_{out}} \setminus \mathcal{C}.\mathbf{Q_{in}}$ hold messages consumed by the environment, and are denoted $\mathcal{E}.\mathbf{Q_{in}}$. A *transition of the environment* modifies the queues in $\mathcal{E}.\mathbf{Q_{in}} \cup \mathcal{E}.\mathbf{Q_{out}}$ by nondeterministically removing first messages from the queues in $\mathcal{E}.\mathbf{Q_{in}}$ and enqueuing new messages in the queues in $\mathcal{E}.\mathbf{Q_{out}}$. We assume that in each run the tuples enqueued in environment transitions use values from some finite domain. A *run* of $\mathcal{C}$ is defined by allowing, in addition to regular moves of peers in $\mathcal{C}$, non-deterministically interleaved transitions of the environment. These are detected by a special propositional state $move_{\mathcal{E}}$ that is true whenever a transition of the environment occurs. We omit the formal definition, which is an extension of Definition 2.6.

We next formalize the modular verification problem. An *environment* specification (spec) for $\mathcal{C}$ is an LTL-FO formula over $\mathcal{C}.\mathbf{Q_{in}} \,\Delta\, \mathcal{C}.\mathbf{Q_{out}}$. Thus, an environment spec describes the input-output behavior of the outside peers as temporal connections between messages they receive and send.

Example **5.1** We are interested in verifying that $\mathcal{O}$ from Example 2.2 satisfies property (11) from Example 3.2 provided that the credit reporting agency replies to credit inquiries, and moreover returns only credit categories from a pre-defined list (ranging from "*poor*" to "*excellent*"). This is expressed as an environment specification as follows:

$$\begin{gathered}\mathbf{G}\ \forall ssn\ [?\mathbf{getRating}(ssn) \\ \rightarrow \\ (!\mathbf{rating}(ssn, \text{“}poor\text{”}) \vee !\mathbf{rating}(ssn, \text{“}fair\text{”}) \\ \vee !\mathbf{rating}(ssn, \text{“}good\text{”}) \vee !\mathbf{rating}(ssn, \text{“}excellent\text{”}))]\end{gathered} \tag{12}$$

□

Intuitively, it is natural to interpret the specification of an environment with observer-at-source semantics, independently of the properties of the channels connecting the composition $\mathcal{C}$ with its environment. In other words, if Q is an output of the environment, an atom $Q(\bar{x})$ used in the description of the environment is true at some point in a run iff message $Q(\bar{x})$ is sent at that point by the environment. However, recall that in the context of bounded queues and lossy channels, our observer-at-recipient semantics ignores dropped messages, so $Q(\bar{x})$ is taken to be the last enqueued message in the queue for Q. Thus, if the environment sends $Q(\bar{x})$ at step i, the only fact observable at the recipient is that, if a message is received at step $i+1$ in queue Q, then it has to equal $Q(\bar{x})$. To correctly interpret environment specs in the context of bounded queues and lossy channels with our observer-at-recipient semantics, we translate their specification as follows. Let ψ be an environment spec. The *observer-at-recipient* translation of ψ is the formula ψ_r obtained by replacing in ψ each atom $Q(\bar{x})$ where $Q \in \mathcal{E}.\mathbf{Q_{out}}$ by $\mathbf{X}(received_Q \rightarrow Q(\bar{x}))$, where $received_Q$ is a new propositional state that holds at step i iff the queue for Q received a new message between step $i-1$ and i ($received_Q$ holds either if the length of queue Q increased from step $i-1$ to step i, or if a message was read from Q at step $i-1$ and the length of Q stayed the same; this can be defined in terms of states and inputs already available, so $received_Q$ is simply a convenient shorthand).

Example **5.2** The observer-at-recipient translation of the environment in Example 5.1 is

$$\begin{gathered}\mathbf{G}\ \forall ssn\ [?\mathbf{getRating}(ssn) \rightarrow \\ \mathbf{X}(received_{!\mathbf{rating}} \rightarrow \\ (!\mathbf{rating}(ssn, \text{“}poor\text{”}) \vee !\mathbf{rating}(ssn, \text{“}fair\text{”}) \\ \vee !\mathbf{rating}(ssn, \text{“}good\text{”}) \vee !\mathbf{rating}(ssn, \text{“}excellent\text{”})))]\end{gathered}$$

(after combining several implications with $received_{!\mathbf{rating}}$ on the left-hand side). □

The soundness of the observer-at-recipient translation can be shown formally as follows: (i) define observer-at-source runs of the environment recording the consumed and generated messages at each transition, (ii) define satisfaction of an environment spec ψ by observer-at-source runs of the environment, (iii) define the set of observer-at-recipient runs corresponding to observer-at-source runs, and (iv) show that for each environment spec ψ, the observer-at-recipient runs corresponding to the observer-at-source runs satisfying ψ are precisely those satisfying ψ_r. We omit the straightforward details here.

We are now ready to define satisfaction of a property φ under environment spec ψ. Since in a run of $\mathcal{C}$ the environment transitions are interleaved with transitions of peers in $\mathcal{C}$, the property ψ describing runs of the environment must be relaxed to take into account the interleaved transitions. Intuitively, this is done by considering only configurations where $move_{\mathcal{E}}$ is true. In detail, for a propositional state α, we denote by $\mathbf{X}^{\alpha}$ a temporal operator whose semantics is the following:

- $\rho_{\geq j} \models \mathbf{X}^{\alpha}\varphi$ iff $\rho_{\geq i} \models \varphi$, where $i = min\{m \mid m > j, \rho_m \models \alpha\}$;
- $\rho_{\geq j} \models \xi_1 \mathbf{U}^{\alpha} \xi_2$ iff $\exists k \geq j$ such that $\rho_k \models \alpha$ and $\rho_{\geq k} \models \xi_2$ and $\rho_{\geq m} \models \xi_1$ for every m such that $j \leq m < k$ and $\rho_m \models \alpha$.

It is clear that $\mathbf{X}^{\alpha}$ and $\mathbf{U}^{\alpha}$ can be simulated with usual LTL operators.

Definition 5.3. *Let $\mathcal{C}$ be an open composition, φ an LTL-FO formula over the schema of $\mathcal{C}$, and ψ an environment spec for $\mathcal{C}$. Let $\alpha = move_{\mathcal{E}}$ and $\bar{\psi}$ be obtained by replacing in ψ each occurrence of the* $\mathbf{X}$ *and* $\mathbf{U}$ *operators with* $\mathbf{X}^{\alpha}$ *and* $\mathbf{U}^{\alpha}$. *Let $\bar{\psi}_r$ be the observer-at-recipient translation of $\bar{\psi}$. Then $\mathcal{C}$ satisfies φ under environment spec ψ (denoted*

$\mathcal{C} \models_\psi \varphi$) iff every run ρ of $\mathcal{C}$ that satisfies $\bar{\psi}_r$ also satisfies φ.

Note that the order of the two translations, first from ψ to $\bar{\psi}$ and then from $\bar{\psi}$ to $\bar{\psi}_r$, is important and cannot be switched. Indeed, the translation from $\bar{\psi}$ to $\bar{\psi}_r$ introduces $\mathbf{X}$ operators that must not be replaced by $\mathbf{X}^\alpha$.

Towards stating our decidability result for modular verification, we recall from Section 3 that an LTL-FO formula allows no temporal operators to appear in the scope of any quantifier. An LTL-FO sentence relaxes this restriction, being obtained by universally quantifying the free variables of an LTL-FO formula. In contrast, we say that an LTL-FO sentence is *strictly input-bounded* if no temporal operators occur in the scope of quantifiers. The environment spec (12) is strictly input-bounded.

THEOREM 5.4. *It is decidable whether an input-bounded open composition $\mathcal{C}$ with bounded queues and lossy channels satisfies property φ under environment spec ψ, where φ is an input-bounded LTL-FO sentence and ψ is a strictly input-bounded LTL-FO sentence over $\mathcal{C}.\mathbf{Q}^{\mathbf{f}}_{\mathbf{in}} \Delta \mathcal{C}.\mathbf{Q}^{\mathbf{f}}_{\mathbf{out}}$. Moreover, the problem is* PSPACE*-complete for schemas of bounded arity and* EXPSPACE *otherwise.*

As it turns out, the strictness restriction is essential; removing it leads to undecidability.

THEOREM 5.5. *It is undecidable whether an input-bounded open composition $\mathcal{C}$ with bounded queues and lossy channels satisfies input-bounded property φ under input-bounded yet* non-strict *environment specifications.*

It is easy to see that strictly input-bounded LTL-FO environment specs can be expressed by input-bounded conversation protocols. It turns out that the proof of Theorem 5.4 above adapts to obtain decidability when the environment spec is given instead by an input-bounded conversation protocol with observer-at-recipient semantics.

6. RELATED WORK

In the finite-state case, it was shown in prior work that verification of communicating finite-state machines (CFSM) is undecidable for *unbounded, perfect* queues [6],and for *unbounded, lossy* queues [2]. The CFSM model is a special case of ours in which all schemas are propositional and there is no user input or database.

The body of work on compositions of communicating finite-state Web Services (sometimes called *e-compositions*) is surveyed in [22, 23, 24]. We mention a few projects here. [15] verifies that synchronous finite-state mediated composite services specified in the standard BPEL language [10] implement a Message Sequence Chart specification. The verification is performed by compiling the sequence charts into the Finite State Process notation (FSP), and invoking a propositional model checker from the LTSA toolkit. [26] proposes an approach to the verification and automated composition of finite-state web services specified using the DAML-S standard [11]. The verified properties are propositional, abstracting from the data values. They pertain to safety, liveness and deadlocks, all of which are expressible in LTL. [25] is concerned with verifying a given finite-state web service flow specified in the standard WSFL [29] by using the explicit state model checker SPIN [20]. The properties are expressed in LTL (abstracting from data content).

The line of work in [18, 16] takes into account the contents of the exchanged messages and thus transcends the purely propositional composition models described above (but assumes a pre-defined finite domain for the values, which reduces the problem to a finite-state setting). Peers are specified using finite-state automata whose transitions are guarded by boolean formulas involving the message contents. Properties are expressed in LTL. This is a particular case of the framework presented in this paper, with finite domain, no database, no user input and no nested queues (but perfect bounded flat queues). The emphasis is not on mapping the verification boundaries, but on developing a versatile architecture allowing the exchange of XML messages without being tied to any particular standard, as well as sufficient conditions to delegate the verification task to the off-the-shelf finite-state model checker SPIN [20].

Recently,[5] has proposed a model of compositions of peers with underlying databases. The model corresponds to a particular case of the one we present here, with no user input, no nested queues, perfect flat queues, and database access restricted to key lookup only, so that at most one tuple is retrieved or updated at any given time. [5] does not address verification, focusing on automatic synthesis of a desired Web Service by "gluing together" an existing set of services.

7. CONCLUSIONS

We have studied the verification properties of compositions of data-driven peers communicating asynchronously by message exchange. We treat data values as first-class citizens, specifying each peer's behavior with queries against its configuration. This leads to infinite-state compositions (since the database is not fixed in advance), and constitutes a departure from classical work on verification of communicating finite-state machines or, more recently, verification of finite-state Web Service compositions.

We delineate the boundaries of verification decidability by exploring a wide range of communication semantics and classes of composition and property specifications. We also consider modular verification of partially specified compositions when only the input-output behavior of their environment is known. We identify a practically appealing and fairly tight class of specifications for which verification is decidable in PSPACE (for fixed database arity), which is no worse than LTL verification of finite-state machines. Our favorable prior experiments on individual peer verification lead us to expect similar results for compositions.

8. REFERENCES

[1] Wave demo. Available at `http://www.cs.ucsd.edu/~lsui/project`.

[2] P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.

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

[4] S. Abiteboul, L. Herr, and J. V. den Bussche. Temporal versus first-order logic to query temporal databases. In *Proc. ACM PODS*, pages 49–57, 1996.

[5] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, 1983.

[7] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing data-intensive Web applications.* Morgan-Kaufmann, 2002.

[8] A. K. Chandra and M. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comp.*, 14(3):671–677, 1985.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.

[10] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for Web services. http://dev2dev.bea.com/techtrack/BPEL4WS.jsp.

[11] DAML-S Coalition (A. Ankolekar et al).DAML-S: Web service description for the semantic Web. In *The Semantic Web - ISWC*, pages 348–363, 2002.

[12] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD Conference*, 2005.

[13] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of interactive web services. In *ACM Symposium on Principles of Database Systems (PODS)*, 2004.

[14] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

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

[16] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL Web Services. In *World Wide Web Conference (WWW)*, 2004.

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

[18] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. In *IEEE International Conference on Web Services (ICWS)*, 2004.

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

[20] G. Holzmann. *The Spin Model Checker – Primer and Reference Manual.* Addison-Wesley, 2003.

[21] R. Hull. Web services composition: A story of models, automata, and logics. In *IEEE International Conference on Services Computing (SCC)*, 2005.

[22] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a look behind the curtain. In *PODS*, 2003.

[23] R. Hull and J. Su. Tools for design of composite web services. In *SIGMOD Conference*, 2004.

[24] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2):86–95, 2005.

[25] S. Nakajima. Verification of web service flows with model-checking techniques. In *Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, 2002.

[26] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of Web services. In *Proc. WWW*, pages 77–88, 2002.

[27] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS.*, 66(1):40–65, 2003. Extended abstract in PODS 2000.

[28] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symp. on Logic in Computer Science*, 1986.

[29] Web Services Flow Language(WSFL 1.0), 2001. http://www-3.ibm.com/ software/ solutions /webservices/pdf/WSFL.pdf.

APPENDIX

A. SOME PROOFS

PROOF. **(of Theorem 3.4)** The PSPACE-hardness follows directly from Theorem 3.5 in [13], which shows PSPACE-hardness even for a single peer. We prove the upper bound. For the sake of presentation simplicity, we first present the proof for 1-bounded queues, then sketch the extension to k-bounded queues.

Essentially, we reduce in PTIME the verification of input-bounded compositions with k-bounded queues and lossy channels to verification of isolated input-bounded Web services with no message queues and k-lookback. Recall that a peer has k-lookback if it can refer at every step, for each input I, to the j-th previous non-empty input to I (denoted by $prev_I^j$), for $1 \leq j \leq k$. The reduction in conjunction with Lemma 3.5 establishes decidability of verification and its complexity.

Specifically, consider an input-bounded composition $\mathcal{C}$ with k-bounded queues and lossy channels, and an input-bounded property φ. We construct an input-bounded Web service $\mathcal{W}_c$ with no message queues and k-lookback, as well as an input-bounded property ψ such that $\mathcal{C} \models \varphi$ iff $\mathcal{W}_c \models \psi$. The reduction features the following key ingredients:

- We need to model the fact that at each step precisely one non-deterministically chosen peer gets to move. We do so by having $\mathcal{W}_c$ generate as input options the names of all peers. A dummy user will pick the peer to move next, and $\mathcal{W}_c$ executes the move of the chosen peer.
- We model each nested queue by a state, and the sending/receiving operations as updates of this state. The state is flushed after each reception. The non-deterministic decision on whether the message is lost in transition is made using a propositional input set by a dummy user.
- Notice that the sending of a tuple along a flat queue is subject to two sources of non-determinism: the first pertains to picking one among the multiple candidate messages generated by the rule, the second to deciding whether the sent message is lost in transmission. We model the combined non-determinism by turning the send rule into an input rule. The input rule generates the message candidates among which at most one is picked by a dummy user.
- Since a send rule r_s may depend on the current input, the input rule modeling it, r_i, must be evaluated after the current input has been chosen. To deal with this one-step timing mismatch, we introduce for each input I another input early_I whose function is to provide at each step the value of I at the next step. Then $prev_{early_I}$ can be used in the current step in the evaluation of r_i.

There are two means available for achieving the above simulation. One is the definition of $\mathcal{W}_c$ itself. However, $\mathcal{W}_c$ may generate some runs that do not correspond to runs of $\mathcal{C}$ and that need to be filtered out. To do this, we use the definition of the property ψ. Specifically, we define ψ to be of the form $\psi_0 \rightarrow \varphi^*$ where φ^* corresponds to φ and ψ_0 is an input-bounded LTL-FO sentence satisfied only by the runs of $\mathcal{W}_c$ that correspond to runs of $\mathcal{C}$. For example, in the use of inputs to simulate the choice of peer for each move, the dummy user may choose to pick no peer. This generates a run of $\mathcal{W}_c$ that does not correspond to a run of $\mathcal{C}$. However, this run can be filtered out by ψ_0 by requiring all choices for this input to be non-empty throughout the run. Another example concerns the early_I relations associated to each input I. In the definition of $\mathcal{W}_c$, the input options for early_I consist of the cross product of the active domain, so early_I has no connection to I. However, ψ_0 can be used to select only those runs where early_I contains at each step the value of I at the next step (as will be seen, the first step requires special treatment).

Formally, $\mathcal{W}_c$ is defined as follows.

$\mathcal{W}_c.\mathbf{Q_{in}} = \emptyset$, $\mathcal{W}_c.\mathbf{Q_{out}} = \emptyset$,

$\mathcal{W}_c.\mathbf{D}$ contains

- the database relations of all peers, $\mathcal{C}.\mathbf{D}$.
- for each $\mathcal{W} \in \mathcal{C}$, a constant $c_{\mathcal{W}}$.
- for each input I of arity m in $\mathcal{C}.\mathbf{I}$, m constants $c_I^1, \ldots, c_I^m$ and a proposition $I_\emptyset$.

$\mathcal{W}_c.\mathbf{I}$ contains

- the inputs of all peers, $\mathcal{C}.\mathbf{I}$,
- a unary input move,
- for each input I in $\mathcal{C}.\mathbf{I}$, an input early_I of the same arity,
- for each $R \in \mathcal{C}.\mathbf{Q^f_{out}}$, an input I_R of same arity as R.
- for each $R \in \mathcal{C}.\mathbf{Q^n_{out}}$, a propositional input lost_R.

$\mathcal{W}_c.\mathbf{S}$ contains

- the states of all peers, $\mathcal{C}.\mathbf{S}$,
- a state S_R of arity m for each m-ary $R \in \mathcal{C}.\mathbf{Q^n_{out}}$;
- a propositional state full_Q for each queue $Q \in \mathcal{C}.\mathbf{Q_{in}}$;
- a propositional state $\mathit{empty\text{-}early}_I$ for each input in $\mathcal{C}.\mathbf{I}$.
- a propositional state $\mathit{notfirst}$ used to identify the initial configuration.

We define the rules of $\mathcal{W}_c$ next. In the following we use for any FO formula φ the shorthand φ^* to denote the formula obtained by simultaneously substituting in φ each occurrence of an atom $Q(\bar{x})$ where $Q \in \mathcal{W}.\mathbf{Q^f_{in}}$ with $\mathit{full}_Q \wedge \mathit{prev}_{I_Q}(\bar{x})$, and each occurrence of a nested in-queue symbol $Q \in \mathcal{W}.\mathbf{Q^n_{in}}$ with the state S_Q.

$\mathcal{W}_c.\mathcal{R}$ contains:

1. the input rules
$$\mathit{Options}_{\mathsf{move}}(x) \leftarrow \bigvee_{\mathcal{W} \in \mathcal{C}} x = c_{\mathcal{W}},$$
for each $R \in \mathcal{C}.\mathbf{Q^n_{out}}$,
$$\mathit{Options}_{\mathsf{lost}_R} \leftarrow \mathit{true},$$
and for each input early_I of arity m the rule
$$\mathit{Options}_{\mathsf{early}_I} \leftarrow \xi_{adom^m}$$
where ξ_{adom^m} stands for an $\exists^*$FO formula defining the m-ary cross product of the active domain.
2. for each $\mathcal{W} \in \mathcal{C}$ and each input rule $\mathit{Options}_I(\bar{x}) \leftarrow \varphi_I(\bar{x})$ in $\mathcal{W}.\mathcal{R}$, the input rule
$$\mathit{Options}_I(\bar{x}) \leftarrow \mathsf{move}(c_W) \wedge \varphi_I^*(\bar{x});$$
3. for each peer $\mathcal{W} \in \mathcal{C}$, each $S \in \mathcal{W}.\mathbf{S}$ and each state rule $(\neg)S(\bar{x}) \leftarrow \varphi_S(\bar{x})$ in $\mathcal{W}.\mathcal{R}$, the state rule
$$(\neg)S(\bar{x}) \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \varphi_S^*(\bar{x});$$
4. for each input $I \in \mathcal{C}.\mathbf{I}$ the state rules
$$\neg \mathit{empty\text{-}early}_I \leftarrow \exists \bar{x}\ \mathsf{early}_I(\bar{x}),$$
$$\mathit{empty\text{-}early}_I \leftarrow \neg\exists \bar{x}\ \mathsf{early}_I(\bar{x})$$
5. the state rule
$$\mathit{notfirst} \leftarrow \neg \mathit{notfirst}$$
6. for each peer $\mathcal{W} \in \mathcal{C}$ and each action rule $A(\bar{x}) \leftarrow \varphi_A(\bar{x})$ in $\mathcal{W}.\mathcal{R}$, the action rule
$$A(\bar{x}) \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \varphi_A^*(\bar{x});$$
7. for each $\mathcal{W} \in \mathcal{C}$, each $Q \in \mathcal{W}.\mathbf{Q^n_{out}}$ and each send rule $Q(\bar{x}) \leftarrow \varphi_Q(\bar{x})$ in $\mathcal{W}.\mathcal{R}$, the state rules
$$S_Q(\bar{x}) \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \neg\, \mathit{full}_Q \wedge \neg\, \mathsf{lost}_Q \wedge \varphi_Q^*(\bar{x});$$
$$\neg\, S_Q(\bar{x}) \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \neg\, \mathit{full}_Q \wedge \neg\, \mathsf{lost}_Q \wedge S_Q(\bar{x});$$
$$\mathit{full}_Q \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \neg\, \mathsf{lost}_Q;$$
8. for each peer $\mathcal{W} \in \mathcal{C}$ and each $Q \in \mathcal{W}.\mathbf{Q_{in}}$, the state rules
$$\neg\, \mathit{full}_Q \leftarrow \mathsf{move}(c_{\mathcal{W}}),$$
$$\mathit{empty}_Q \leftarrow \neg \mathit{full}_Q,$$
$$\neg \mathit{empty}_Q \leftarrow \mathit{full}_Q;$$
9. for each $\mathcal{W} \in \mathcal{C}$, each $Q \in \mathcal{W}.\mathbf{Q^f_{out}}$ and each send rule $Q(\bar{x}) \leftarrow \varphi_Q(\bar{x})$ in $\mathcal{W}.\mathcal{R}$, the input rule
$$\mathit{Options}_Q(\bar{x}) \leftarrow \mathsf{move}(c_{\mathcal{W}}) \wedge \neg\, \mathit{full}_Q \wedge (\varphi_{notfirst} \vee \varphi_{first})$$
where
   - $\varphi_{notfirst}$ is obtained from φ_Q by substituting each occurrence of an atom $I(\bar{x})$ for input $I \in \mathcal{C}.\mathbf{I}$ with $\mathit{prev}_{early_I}(\bar{x}) \wedge \neg \mathit{empty\text{-}early}_I$, replacing each occurrence of a nested in-queue symbol $R \in \mathcal{W}.\mathbf{Q^n_{in}}$ with the state S_R and taking the conjunction with $\mathit{notfirst}$,
   - φ_{first} is obtained by substituting each occurrence of an atom $I(x_1, \ldots, x_m)$ for input $I \in \mathcal{C}.\mathbf{I}$ with $x_1 = c_I^1 \wedge \ldots \wedge x_m = c_I^m \wedge \neg I_\emptyset$, replacing each occurrence of a nested in-queue symbol $R \in \mathcal{W}.\mathbf{Q^n_{in}}$ with the state S_R, and taking the conjunction with $\neg \mathit{notfirst}$.

   Additionally, we have the following state rule:
$$\mathit{full}_Q \leftarrow \exists \bar{x} Q(\bar{x})$$

Intuitively, the simulation of $\mathcal{C}$ by $\mathcal{W}_c$ works as follows. At each step, a dummy user picks the peer to move next using the input move. (rule 1.): if the user picks $c_{\mathcal{W}}$, then peer $\mathcal{W}$ moves next. The definition of $\mathcal{W}_c$ cannot require the user to pick some peer, but this will be ensured by the "filter" φ_0 used in defining the property ψ to be verified. The flat-queue send rules are emulated by input rules (rules 9.) which generate the message candidates from which a dummy user picks the one to be actually sent. Note how current inputs I, unavailable to these rules, are replaced by the inputs early_I (or by database constants $c_I^1, \ldots, c_I^m$ in the first step). The dummy user may choose to pick no input, thus modeling the loss of a message. $\mathcal{W}_c$ evaluates the input, state and action rules of peer $\mathcal{W}$ (via rules 2., 3. and 6., respectively), if $\mathcal{W}$ is currently scheduled to move. $\mathcal{W}_c$ emulates $\mathcal{W}$'s sending into nested queues by appropriate state rules (7.). These overwrite the state S_Q modeling queue Q and set the $full_Q$ flag, unless the input lost_Q indicates that the message is lost in transmission. The $full_Q$ flag is reset upon reception (8.).

Now let us turn to the definition of the formula ψ, which, as discussed earlier, is of the form $\psi_0 \rightarrow \varphi^*$. The formula ψ_0 consists of the conjunction of the following:

1. $\mathbf{G}\ \exists x\ \mathsf{move}(x)$,
2. for each input I in $\mathcal{C}.\mathbf{I}$ of arity m,

$$\forall x_1 \ldots \forall x_m (I(x_1, \ldots, x_m) \rightarrow x_1 = c_I^1 \wedge \ldots \wedge x_m = c_I^m)$$

$$\wedge\ I_\emptyset \leftrightarrow \neg \exists x_1 \ldots \exists x_m I(x_1, \ldots, x_m),$$

$$\mathbf{XG}\ \textit{empty-early}_I \leftrightarrow \neg \exists \bar{x} I(\bar{x}),$$

$$\mathbf{XG}\ \forall \bar{x}((\mathsf{prev}_{early_I}(\bar{x}) \wedge \neg \textit{empty-early}_I) \leftrightarrow I(\bar{x}));$$

Note that the sentence ψ_0 is input-bounded. In ψ_0, the formula (2) ensures the synchronization between early_I and I after the first step, and between I and the database constants $c_I^1, \ldots, c_I^m$ at the first step, with $\textit{empty-early}_I$ and $I_\emptyset$ flagging empty inputs. This allows flat message rules requiring I to use instead early_I in the simulation everywhere except at the first step, where $c_1, \ldots, c_m$ are used instead. Clearly, each run of $\mathcal{W}_c$ that satisfies ψ_0 corresponds to a run of $\mathcal{C}$, where flat and nested queues are simulated by the corresponding inputs and states. Finally, let $\psi = \psi_o \rightarrow \varphi^*$. Clearly, $\mathcal{W}_c$ and ψ are both input-bounded and $\mathcal{C} \models \varphi$ iff $\mathcal{W}_c \models \psi$.

The above construction can be easily extended to the case of k-bounded queues for $k > 1$. Flat messages are represented as before by inputs, and a flat queue of size k by the k previous non-empty values of the input (requiring the k-lookback capability). Nested messages are represented by states, and a queue of k nested messages by k states corresponding to the up to k messages in the queue. Additionally, some bookkeeping is needed to keep track of the number of messages currently in each queue, and to flag empty and full queues. This can be easily done using additional propositional states. □

PROOF. (**of Theorem 3.7**) We reduce from the *Post Correspondence Problem (PCP)*. Consider a PCP instance, i.e. two sequences of length n: $\{u_i\}_{1 \leq i \leq n}, \{v_i\}_{1 \leq i \leq n}$, where all u_i, v_j are non-empty words over the alphabet $\{0, 1\}$. A *solution* to $\mathcal{P}$ is a finite non-empty sequence $\sigma \in [1, \ldots, n]^*$ such that the two strings obtained by concatenating $u_{\sigma(1)} u_{\sigma(2)} \ldots u_{\sigma(k)}$ and $v_{\sigma(1)} v_{\sigma(2)} \ldots v_{\sigma(k)}$ are identical ($\sigma(i)$ is the element at position i in σ). We say that these strings are *generated* by the solution σ. We construct a composition $\mathcal{C}$ and a property φ such that $\mathcal{P}$ has a solution iff $\mathcal{C} \not\models \varphi$.

The composition simulates the search for a PCP solution as follows. $\mathcal{C}$ contains two peers, a searcher $\mathcal{W}_s$ and a checker $\mathcal{W}_c$. The local database of $\mathcal{W}_s$ encodes a finite string θ intended to correspond to the string generated by a solution of $\mathcal{P}$. $\mathcal{W}_s$ non-deterministically picks a sequence of indexes from $[1, \ldots, n]$ (by repeatedly asking an external user to pick an input among the options $[1, \ldots, n]$). Upon picking some index i, $\mathcal{W}_s$ tries to match the corresponding words u_i and v_i in parallel against θ, by maintaining two cursors U and V on θ, as well as a cursor on u_i and a cursor on v_i. The cursors advance in lock-step, being incremented only if they point to the same character. Initially, U and V start from the first position in θ. The property φ is satisfied only if for all j, upon finishing to fully match u_j and v_j, U and V never meet on θ. It is easy to see that, if the local database of $\mathcal{W}_s$ encodes a string θ, a run of the composition violates φ if and only if the sequence of indexes picked by $\mathcal{W}_s$ is a solution to $\mathcal{P}$, which generates a prefix of θ.

θ is encoded using two binary database relations, $\underline{\mathsf{chain}}(s, t)$ (intended to contain as a subgraph a chain of directed $s \rightarrow t$ edges) and $\underline{\mathsf{char}}(i, c)$ (intended to label each node i in the chain with a character $c \in \{0, 1\}$). We will enforce that $\underline{\mathsf{chain}}(s, t)$ satisfies the functional dependencies (FDs) $s \rightarrow t$ and $t \rightarrow s$ and $\underline{\mathsf{char}}$ satisfies the FD $i \rightarrow c$. The FDs on $\underline{\mathsf{chain}}$ ensure that nodes have in-degree and out-degree one, so $\underline{\mathsf{chain}}$ is a union of disjoint cycles and chains. To ensure that the cursors U and V progress along the same path without revisiting any node, we enforce that they start from the same position, a special node '\$', and never return to '\$'. The FD on $\underline{\mathsf{char}}$ will ensure that indexes are labeled uniquely, and the rules ensure that the labels are in $\{0, 1\}$ (the fact that 0 and 1 are distinct constants is stated in the property).

To detect violations of the FDs, $\mathcal{W}_s$ sends a (flat) propositional message along queue **viol** to $\mathcal{W}_c$, set to true by FD violations (if any). Since flat queues are perfect, $\mathcal{W}_c$ receives a message iff the FDs are violated. The property will check that no violation message is received.

In detail, the schema of $\mathcal{W}_s$ consists of

- $\mathcal{W}_s.\mathbf{D} = \{\underline{\mathsf{chain}}(s, t),\ \underline{\mathsf{char}}(i, c),\ '\$', 0, 1\}$ as described above ('\$', 0, and 1 are constants);

- $\mathcal{W}_s.\mathbf{I} = \{\mathsf{I}(i), \mathsf{U}(x), \mathsf{V}(x)\}$. Intuitively, the user provides his pick of a word index in I, and U and V are the cursors on θ. The options provided to the user contain the immediate successors in $\underline{\mathsf{chain}}$ of the cursors at the previous input $prev_{\mathsf{U}}, prev_{\mathsf{V}}$. Of course, there is at most one successor if the FDs on $\underline{\mathsf{chain}}$ holds.

- $\mathcal{W}_s.\mathbf{S}$ contains the following propositional states:

  - for each $1 \leq i \leq n$, each $1 \leq j \leq |u_i|$ and each $1 \leq k \leq |v_i|$, state U_i^j and state V_i^k (these play the role of cursors in the u_i and v_i words);

  - state $done_u$, set to true only when a full u_i word is matched; $begun_u$ which, when set to false, signals that the matching of u_i words has not yet begun; similarly, states $done_v$ and $begun_v$.

- $\mathcal{W}_s.\mathbf{A} = \mathcal{W}_s.\mathbf{Q_{in}} = \mathcal{W}_s.\mathbf{Q^n_{out}} = \emptyset$;

- $\mathcal{W}_s.\mathbf{Q^f_{out}} = \{\mathbf{viol}\}$, where **viol** is propositional.

$\mathcal{W}_s$ contains

- the input rules

$$\begin{array}{rcl} Options_{\mathsf{I}}(i) & \leftarrow & (i = 1 \vee i = 2 \vee \ldots \vee i = n) \\ & \wedge & (\neg begun_u \wedge \neg begun_v \vee done_u \wedge done_v) \\ Options_{\mathsf{U}}(t) & \leftarrow & (\neg begun_u \wedge t =' \$') \\ & \vee & begun_u \wedge \neg done_u \wedge \\ & & \exists s \exists c\ prev_{\mathsf{U}}(s) \wedge \underline{\mathsf{chain}}(s,t) \wedge t \neq' \$' \\ & & \wedge \underline{\mathsf{char}}(t,c) \\ & & \wedge (\bigvee_{i,j} prev_{\mathsf{I}}(i) \wedge c = u_i(j) \wedge U_i^j) \\ Options_{\mathsf{V}}(t) & \leftarrow & (\neg begun_v \wedge t =' \$') \\ & \vee & begun_v \wedge \neg done_v \wedge \\ & & \exists s \exists c\ prev_{\mathsf{V}}(s) \wedge \underline{\mathsf{chain}}(s,t) \wedge t \neq' \$' \\ & & \wedge \underline{\mathsf{char}}(t,c) \\ & & \wedge (\bigvee_{i,k} prev_{\mathsf{I}}(i) \wedge c = v_i(k) \wedge V_i^k)) \end{array}$$

- the state rules

$$\begin{array}{rcl} begun_u & \leftarrow & \neg begun_u \wedge \exists t\ \mathsf{U}(t) \\ begun_v & \leftarrow & \neg begun_v \wedge \exists t\ \mathsf{V}(t) \\ done_u & \leftarrow & \exists t\ \mathsf{U}(t) \wedge (\bigvee_{i=1}^{n} U_i^{|u_i|-1}) \\ \neg done_u & \leftarrow & done_u \wedge \exists x\ \mathsf{I}(x) \\ done_v & \leftarrow & \exists t\ \mathsf{V}(t) \wedge (\bigvee_{i=1}^{n} V_i^{|v_i|-1}) \\ \neg done_v & \leftarrow & done_v \wedge \exists x\ \mathsf{I}(x) \end{array}$$

Moreover, for $1 \leq i \leq n$,

$$\begin{array}{rcll} U_i^1 & \leftarrow & \mathsf{I}(i) & \\ U_i^j & \leftarrow & U_i^{j-1} \wedge \exists t\ \mathsf{U}(t) & \text{for } 1 < j \leq |u_i| \\ \neg U_i^j & \leftarrow & U_i^j \wedge \exists t\ \mathsf{U}(t) & \text{for } 1 \leq j \leq |u_i| \\ V_i^1 & \leftarrow & \mathsf{I}(i) & \\ V_i^j & \leftarrow & V_i^{j-1} \wedge \exists t\ \mathsf{V}(t) & \text{for } 1 < j \leq |v_i| \\ \neg V_i^j & \leftarrow & V_i^j \wedge \exists t\ \mathsf{V}(t) & \text{for } 1 \leq j \leq |v_i| \end{array}$$

- the send rule

$$\begin{array}{rcl} !\mathbf{viol}() & \leftarrow & \exists x \exists y_1 \exists y_2\ ((\underline{\mathsf{chain}}(x,y_1) \wedge \underline{\mathsf{chain}}(x,y_2)) \vee \\ & & (\underline{\mathsf{chain}}(y_1,x) \wedge \underline{\mathsf{chain}}(y_2,x)) \vee \\ & & (\underline{\mathsf{char}}(x,y_1) \wedge \underline{\mathsf{char}}(x,y_2))) \wedge y_1 \neq y_2 \end{array}$$

Finally, the property φ is

$$\begin{array}{ll} 0 \neq 1 \wedge & \\ \forall t & \mathbf{G}(\neg \mathcal{W}_c.?\mathbf{viol}) \rightarrow \\ & \mathbf{G}\neg(\mathcal{W}_s.prev_{\mathsf{U}}(t) \wedge \mathcal{W}_s.prev_{\mathsf{V}}(t) \\ & \quad \wedge \mathcal{W}_s.done_u \wedge \mathcal{W}_s.done_v) \end{array} \tag{13}$$

□

Proof. (**of Theorem 3.8**) The proof is by reduction from the Post Correspondence Problem, and it is a variation on the proof of Theorem 3.7. As in the proof of Theorem 3.7, we need to enforce the FDs on relations $\underline{\mathsf{chain}}$ and $\underline{\mathsf{char}}$. We use the same peers $\mathcal{W}_s$ and $\mathcal{W}_c$, but modify the rule for the flat queue **viol** by including in the message the values of x, y_1, y_2 witnessing a violation. Since each violation produces at least two tuples, this leads to a violation of determinism and sets the error state $error_{\mathbf{viol}}$ to true. This in turn can be detected by the property. The rest of the reduction is unchanged. □

Proof. (**of Theorem 3.9**) If perfect flat message channels are allowed, undecidability follows from Theorem 3.7. So, let us assume the flat message channels are lossy. The proof is by reduction of the implication problem for functional and inclusion dependencies, known to be undecidable [8]. Let Δ be a set of FDs and IDs over a relation S, and f an FD over the same relation. We construct an input-bounded composition $\mathcal{C}$ and an input-bounded LTL-FO property φ such that $\Delta \models f$ iff $\mathcal{C} \models \varphi$.

The idea is that the satisfaction of a constraint by relation S can be checked by testing emptiness of queries involving joins or differences of projections of S. For instance, an FD of form $X \rightarrow A$ is satisfied by S if the join $\{(t_1.X, t_1.A, t_2.A) \mid t_1 \in \Pi_{X,A}(S) \wedge t_2 \in \Pi_{X,A}(S) \wedge t_1.X = t_2.X \wedge t_1.A \neq t_2.A\}$ is empty. Similarly, S satisfies ID $|X| \subseteq |Y|$ if the difference $\{t \mid t \in \Pi_X(S) \wedge t \notin \Pi_Y(S)\}$ is empty.

The composition contains two peers, $\mathcal{C} = \{\mathcal{W}_1, \mathcal{W}_2\}$. $\mathcal{W}_1$ has a local database relation R (of same arity as S), from which at each step it sends one (non-deterministically chosen) tuple to $\mathcal{W}_2$ using a flat queue **data**. $\mathcal{W}_1$ also sends a propositional message along the flat queue **done**.

As long as the **done** messages are lost, $\mathcal{W}_2$ receives the incoming **data** tuples one-by-one and accumulates their corresponding projections into local state relations – one state per required projection. At every step, the states hold the projection of the subset of R which was received by $\mathcal{W}_2$. Once the **done** message is received, $\mathcal{W}_2$ sends, for each FD and ID $\sigma \in \Delta \cup \{f\}$, the join, respectively difference of the corresponding projection states into a nested out-queue $\mathbf{viol}_\sigma$. Clearly, $empty(?\mathbf{viol}_\sigma)$ holds at every step in a run only if either **done** drops all messages, or σ is satisfied.

Suppose first the nested queue channels are perfect. The property φ checks that, if the first message of all nested queues corresponding to Δ is empty at every step of the run, so is the first message of the queue $\mathbf{viol}_f$:

$$\varphi\ :=\ \mathbf{G}(\bigwedge_{\sigma \in \Delta} empty(?\mathbf{viol}_\sigma)) \rightarrow \mathbf{G}(empty(?\mathbf{viol}_f)).$$

If the nested channels are lossy, the property uses the built-in states $empty_Q$ signaling emptiness of the queue Q (not to be confused with the new emptiness tests $empty(Q)$ on the *contents* of the first nested message in the queue Q) to filter our runs with lost nested messages:

$$\varphi\ :=\ \mathbf{G}(?\mathbf{done} \rightarrow \mathbf{X}(\bigwedge_{\sigma \in \Delta \cup f} \neg empty_{?\mathbf{viol}_\sigma})) \rightarrow$$

$$[\mathbf{G}(\bigwedge_{\sigma \in \Delta} empty(?\mathbf{viol}_\sigma)) \rightarrow \mathbf{G}(empty(?\mathbf{viol}_f))].$$

Formally, $\mathcal{W}_1$'s schema is given as $\mathcal{W}_1.\mathbf{I} = \mathcal{W}_1.\mathbf{S} = \mathcal{W}_1.\mathbf{A} = \emptyset$, $\mathcal{W}_1.\mathbf{D} = \{R\}$, $\mathcal{W}_1.\mathbf{Q}^{\mathbf{f}}_{\mathbf{out}} = \{\mathbf{data}, \mathbf{done}\}$,
where R, **data** have the same arity as S, and **done** is propositional.

The schema of $\mathcal{W}_2$ is the following: $\mathcal{W}_2.\mathbf{I} = \mathcal{W}_2.\mathbf{D} = \mathcal{W}_2.\mathbf{A} = \emptyset$, $\mathcal{W}_2.\mathbf{Q}^{\mathbf{f}}_{\mathbf{in}} = \{\mathbf{data}, \mathbf{done}\}$, and $\mathcal{W}_2.\mathbf{S}$ contains:

- for each ID σ of the form $[X] \subseteq [Y]$ in Δ, a relation S_X of arity $|X|$ and a relation S_Y of arity $|Y|$;

- for each FD σ of the form $X \to A$ in $\Delta \cup \{f\}$, a relation S^{σ}_{XA} of arity $|X|+1$;

Finally, for each $\sigma \in \Delta \cup \{f\}$, $\mathcal{W}_2.\mathbf{Q^n_{out}}$ contains a relation $\mathbf{viol}_\sigma$.

$\mathcal{W}_1.\mathcal{R}$ contains only two send rules:

$$\begin{array}{rcl} !\mathbf{data}(\bar{x}) & \leftarrow & R(\bar{x}) \\ !\mathbf{done}() & \leftarrow & \mathit{true} \end{array}$$

$\mathcal{W}_2.\mathcal{R}$ consists of:

- for each ID σ of the form $[X] \subseteq [Y]$ in Δ, assuming w.l.o.g. that $R = R[X, Y, Z]$, the state rules

  $$S_X(\bar{x}) \leftarrow \neg ?\mathbf{done} \wedge \exists \bar{y} \exists \bar{z}\ ?\mathbf{data}(\bar{x}, \bar{y}, \bar{z})$$

  and

  $$S_Y(\bar{y}) \leftarrow \neg ?\mathbf{done} \wedge \exists \bar{x} \exists \bar{z}\ ?\mathbf{data}(\bar{x}, \bar{y}, \bar{z})$$

  where $\bar{x}, \bar{y}, \bar{z}$ are tuples of variables of arity $|X|, |Y|, |Z|$, respectively; also the send rule

  $$!\mathbf{viol}_\sigma(\bar{x}) \leftarrow ?\mathbf{done} \wedge S_X(\bar{x}) \wedge \neg S_Y(\bar{x}).$$

- for each FD σ of the form $X \to A$ in $\Delta \cup \{f\}$, assuming w.l.o.g. that $R = R[XA, U]$, the state rule

  $$S^{\sigma}_{XA}(\bar{x}, a) \leftarrow \neg ?\mathbf{done} \wedge \exists \bar{u}\ ?\mathbf{data}(\bar{x}, a, \bar{u})$$

  and the send rule

  $$!\mathbf{viol}_\sigma(\bar{x}, a_1, a_2) \leftarrow ?\mathbf{done} \wedge S^{\sigma}_{XA}(\bar{x}, a_1) \wedge S^{\sigma}_{XA}(\bar{x}, a_2) \wedge a_1 \neq a_2$$

Note that all rules are input-bounded. $\square$

Proof. **(of Theorems 4.2 and 4.5)** The PSPACE-hardness is shown in both cases by an easy reduction from Quantified Boolean Formula, that we omit.

For the upper bounds, note that the proof of Lemma 3.5 (and therefore of Theorem 3.4) actually prove decidability not just for LTL-FO properties, but for protocols $(\Sigma, \mathcal{B}, \{\varphi_\sigma\}_{\sigma \in \Sigma})$ in which φ_σ are expressed over the entire schema of the composition $\mathcal{C}$ (as opposed to only $\mathcal{C}.\mathbf{Q_{out}}$ as required by conversation protocols). Indeed, the proof of Lemma 3.5 proceeds by first compiling an LTL-FO property to a protocol $\mathcal{P}$ and then deciding whether the runs satisfy $\mathcal{P}$. The compilation involves introducing a propositional symbol σ for each maximal FO component of the property φ_σ (a maximal sub-formula containing no temporal operators). The resulting propositional LTL property is then compiled to a Büchi automaton $\mathcal{B}$ using the algorithm of [28].

Consider an input-bounded composition $\mathcal{C}$ with lossy channels and bounded queues, and a data-agnostic conversation protocol $(\Sigma, \mathcal{B})$ where $\Sigma = \mathcal{C}.\mathbf{Q_{out}}$. Recall the simulation of $\mathcal{C}$ by a single peer in the proof of Theorem 3.4. In the resulting peer, the enqueuing of a flat message Q occurs at step i iff the input-bounded sentence $\exists \bar{x}\ \mathsf{Q}(\bar{x})$ holds at step $i-1$. The enqueuing of a nested message $Q \in \mathcal{W}.\mathbf{Q^n_{out}}$ occurs at step i iff $\mathsf{move}(c_{\mathcal{W}}) \wedge \neg \mathsf{lost}_Q \wedge \neg \mathit{full}_Q$ holds at step $i-1$. Note that both $\exists \bar{x}\ \mathsf{Q}(\bar{x})$ and $\mathsf{move}(c_{\mathcal{W}})$ are input bounded. The extension of Theorem 3.4 to properties specified using Büchi automata rather than LTL operators yields the result.

For data-aware conversation protocols, the input-boundedness restriction on the conversation protocol corresponds to input-boundedness of an FO formula expressed only over the $\mathcal{C}.\mathbf{Q_{out}}$ schema. The result follows from the extension of Theorem 3.4 discussed above. $\square$

Proof. **(of Theorem 4.3)** The proof is an easy modification of the proof of Theorem 3.7. Referring to that proof, recall that the flat message **viol** is used to detect violations of the FDs, and the inputs U and V are used as cursors. A solution to the PCP is found if there is a run in which there is no violation and $\exists t(\mathit{prev}_{\mathsf{U}}(t) \wedge \mathit{prev}_{\mathsf{V}}(t) \wedge \mathit{done}_u \wedge \mathit{done}_v)$. Instead of checking this using the property, we define a new flat out-queue **match** in $\mathcal{W}_s$ defined by

$$!\mathbf{match} \leftarrow \exists t(\mathit{prev}_{\mathsf{U}}(t) \wedge \mathit{prev}_{\mathsf{V}}(t) \wedge \mathit{done}_u \wedge \mathit{done}_v).$$

The conversation protocol states that if no **viol** message is ever sent then no **match** message is ever sent. The composition satisfies the protocol under the observer-at-source semantics iff there is no solution to the instance of the PCP. Note that the protocol is data-agnostic. $\square$

Proof. **(of Theorem 4.6)** (i) The proof follows from undecidability results on peers which are communicating finite-state machines (CFSM) with lossy queues holding propositional messages (Abdulla and Jonsson [2]) (ii) The proof is a slight variation of that of Theorem 4.3. Consider the composition constructed there from a given PCP instance, but with perfect flat channel semantics. Consider also the data-agnostic protocol stating that if no **viol** message is ever enqueued then no **match** message is ever enqueued. The protocol is satisfied by the composition iff the PCP instance has no solution. (iii) The proof is again an easy modification of the proof of Theorem 4.3. We replace the flat out-queue **match** in $\mathcal{W}_s$ by a nested unary queue with the same name whose rule is

$$!\mathbf{match}(0) \leftarrow \exists t(\mathit{prev}_{\mathsf{U}}(t) \wedge \mathit{prev}_{\mathsf{V}}(t) \wedge \mathit{done}_u \wedge \mathit{done}_v).$$

We also add to $\mathcal{W}_s.\mathbf{Q_{out}}$ a nested unary message **error** whose rule is

$$!\mathbf{error}(0) \leftarrow \mathit{error}_{\mathbf{viol}}$$

The conversation protocol states that if no message $\mathbf{error}(0)$ is ever sent then no message $\mathbf{match}(0)$ is ever sent. Note that the both the composition and the protocol are input bounded. (iv) For perfect flat messages, the result follows from (ii). For lossy flat messages and lossy or perfect nested messages, the proof is identical to that of Theorem 3.9. $\square$

Proof. **(of Theorem 5.4)** We simulate runs of $\mathcal{C}$ using a single additional input-bounded peer $\mathcal{E}$, that emulates all possible transitions of the environment. Then we reduce the modular verification of $\mathcal{C}$ to standard verification of the closed composition $\mathcal{C} \cup \{\mathcal{E}\}$.

Essentially, $\mathcal{E}$ generates all possible sequences of out-messages from the active domain of its own database. In detail, we reduce $\mathcal{C} \models_\psi \varphi$ to $\mathcal{C} \cup \{\mathcal{E}\} \models \varphi'$, where φ' is input bounded. $\mathcal{E}$ is defined as follows:

- $\mathcal{E}.\mathbf{Q^n_{in}} = \mathcal{C}.\mathbf{Q^n_{out}} \setminus \mathcal{C}.\mathbf{Q^n_{in}}$ $\mathcal{E}.\mathbf{Q^n_{out}} = \mathcal{C}.\mathbf{Q^n_{in}} \setminus \mathcal{C}.\mathbf{Q^n_{out}}$;
- $\mathcal{E}.\mathbf{Q^f_{in}} = \mathcal{C}.\mathbf{Q^f_{out}} \setminus \mathcal{C}.\mathbf{Q^f_{in}}$ and $\mathcal{E}.\mathbf{Q^f_{out}} = \mathcal{C}.\mathbf{Q^f_{in}} \setminus \mathcal{C}.\mathbf{Q^f_{out}}$;
- $\mathcal{E}.\mathbf{I} = \{\mathsf{I}_S, \mathsf{build}_S \mid S \in \mathcal{E}.\mathbf{Q^n_{out}}\}$ where I_S has the same arity as S and build_S is propositional,
- $\mathcal{E}.\mathbf{S} = \{Q_S \mid S \in \mathcal{E}.\mathbf{Q^n_{out}}\} \cup \{Q^+_S \mid S \in \mathcal{E}.\mathbf{Q^n_{in}}\}$, where Q_S and Q^+_S have the same arity as S,
- $\mathcal{E}.\mathbf{A} = \emptyset$,
- $\mathcal{E}.\mathbf{D} = \{D\}$ where D is a unary relation.

We next describe the rules $\mathcal{E}.\mathcal{R}$. The contents of the nested messages in $\mathcal{E}.\mathbf{Q^n_{out}}$ is built non-deterministically over multiple steps, using the inputs I_S whose input options are the

Cartesian product of the active domain of the database, which are accumulated in the states Q_S. Each building up sequence of steps is triggered by the propositional inputs build_S (whose option rules are $\mathsf{build}_S \leftarrow \mathit{true}$) and continue until all of them are false. At that point, the flat and nested messages are sent. For each k-ary $Q \in \mathcal{E}.\mathbf{Q^f_{out}}$, there is a send rule whose body expresses the k-way Cartesian product of the active domain of the database, guarded by the formula $\bigwedge_{S \in \mathcal{E}.\mathbf{Q^n_{out}}} \neg \mathsf{build}_S$ (so that no messages are sent in the building-up phase).

For technical reasons, $\mathcal{E}.\mathcal{R}$ also contains for each $S \in \mathcal{E}.\mathbf{Q^n_{in}}$ the state rules:

$$\neg Q^+_S(\bar{x}) \leftarrow Q^+_S(\bar{x}),$$

$$Q^+_S(\bar{x}) \leftarrow ?\mathbf{S}(\bar{x}).$$

Thus, the state Q^+_S holds the contents of the nested input $?\mathbf{S}$ at the previous step. Clearly $\mathcal{E}$ is input bounded.

The formula φ' is constructed as follows. Let

$$\beta = \bigwedge_{S \in \mathcal{E}.\mathbf{Q^n_{out}}} \neg \mathsf{build}_S.$$

Thus, β is true iff $\mathcal{E}$ is not in the building phase of any nested message. Consider the formula $\bar{\psi}_r$. Let ψ_1 be obtained by replacing in $\bar{\psi}_r$ each temporal operator $\mathbf{X}^\alpha$ and $\mathbf{U}^\alpha$ with $\mathbf{X}^{\alpha\wedge\beta}$ and $\mathbf{U}^{\alpha\wedge\beta}$. Since ψ is strictly input-bounded, its maximal FO components contain no free variables. In the translation to $\bar{\psi}_r$ and then to ψ_1, a maximal FO component θ of ψ is translated to an input-bounded formula θ' using atoms over schema $\mathcal{E}.\mathbf{Q_{in}}$, subformulas of the form $\mathbf{X}(\mathit{received}_Q \rightarrow Q(\bar{x}))$, where $Q \in \mathcal{E}.\mathbf{Q_{out}}$, and no other temporal operators. At this point, the formula θ' may no longer be strictly input bounded, because $\mathbf{X}$ may appear in the range of some quantifier (see Example 5.2). However, the temporal operator $\mathbf{X}$ can be moved in front of the entire formula θ' after replacing each atom $S(\bar{x})$ where $S \in \mathcal{E}.\mathbf{Q_{in}}$ by $\mathsf{prev}_S(\bar{x})$ if $S \in \mathcal{E}.\mathbf{Q^f_{in}}$, and by $Q^+_S(\bar{x})$ if $S \in \mathcal{E}.\mathbf{Q^n_{in}}$. This results in a formula $\mathbf{X}\,\theta''$, where θ'' is an input-bounded FO formula with no free variables. The same can be done for every first-order component of φ. Let the resulting formula be ψ_2. Clearly, ψ_2 is now strictly input bounded. Finally, let φ^* be obtained from φ by replacing each temporal operator $\mathbf{X}$ by $\mathbf{X}^\beta$ and $\mathbf{U}$ by $\mathbf{U}^\beta$. Intuitively, φ^* ignores the building-up steps in moves of $\mathcal{E}$. Let φ' be $(\mathbf{G}\ (\mathbf{F}\beta) \wedge \psi_2) \rightarrow \varphi^*$, where $\mathbf{G}\ (\mathbf{F}\beta)$ states that no building-up phase lasts forever. It is easy to check that φ' is input-bounded over the schema of $\mathcal{C} \cup \{\mathcal{E}\}$, and that $\mathcal{C} \models_\psi \varphi$ iff $\mathcal{C} \cup \{\mathcal{E}\}$ satisfies φ'. Finally, the PSPACE-hardness follows again by an easy reduction from Quantified Boolean Formula, and the upper bounds follow from Theorem 3.4. $\square$

Proof. (**of Theorem 5.5**) The proof is a modification of the proof of Theorem 3.7. The composition now consists of a single peer, $\{\mathcal{W}_{search}\}$, where $\mathcal{W}_{search}$ is a modification of $\mathcal{W}_s$ from Theorem 3.7, in which there is no more local database. The chain is instead provided by the environment. Given two positions s_u, s_v in the chain, $\mathcal{W}_{search}$ sends a request to the environment for the successors t_u, t_v (and characters c_u, respectively c_v) using a flat message $!\mathbf{next}(s_u, s_v)$. The requested data is received in a flat message $?\mathbf{chain}(s_u, t_u, c_u, s_v, t_v, c_v)$.

The state rules are the same as for $\mathcal{W}_s$ from Theorem 3.7, with the addition of

$$\mathit{match} \leftarrow \exists t(\mathit{prev}_{\mathsf{U}}(t) \wedge \mathit{prev}_{\mathsf{V}}(t) \wedge \mathit{done}_u \wedge \mathit{done}_v)$$

which sets the propositional state *match* when a PCP solution is detected.

The input rules are adapted to read the chain information from the queue instead of the database:

$$\begin{array}{rcl}
\mathit{Options}_{\mathsf{I}}(i) & \leftarrow & (i = 1 \vee i = 2 \vee \ldots \vee i = n) \\
& \wedge & (\neg \mathit{begun}_u \wedge \neg \mathit{begun}_v \vee \mathit{done}_u \wedge \mathit{done}_v) \\
\mathit{Options}_{\mathsf{U}}(t_u) & \leftarrow & (\neg \mathit{begun}_u \wedge t_u =' \$') \\
& \vee & \mathit{begun}_u \wedge \neg \mathit{done}_u \wedge \exists s_u \exists c_u \exists s_v \exists c_v\ \mathit{prev}_{\mathsf{U}}(s_u) \\
& & \wedge ?\mathbf{chain}(s_u, t_u, c_u, s_v, t_v, c_v) \wedge t_u \neq' \$' \\
& & \wedge (\bigvee_{i,j} \mathit{prev}_{\mathsf{I}}(i) \wedge c_u = u_i(j) \wedge U^j_i) \\
\mathit{Options}_{\mathsf{V}}(t_v) & \leftarrow & (\neg \mathit{begun}_v \wedge t_v =' \$') \\
& \vee & \mathit{begun}_v \wedge \neg \mathit{done}_v \wedge \exists s_u \exists c_u \exists s_v \exists c_v\ \mathit{prev}_{\mathsf{V}}(s_v) \\
& & \wedge ?\mathbf{chain}(s_u, t_u, c_u, s_v, t_v, c_v) \wedge t_v \neq' \$' \\
& & \wedge (\bigvee_{i,k} \mathit{prev}_{\mathsf{I}}(i) \wedge c_v = v_i(k) \wedge V^k_i))
\end{array}$$

$\mathcal{W}_{search}$ contains the send rule (into the environment)

$$!\mathbf{next}(t_u, t_v) \leftarrow \mathsf{U}(t_u) \wedge \mathsf{V}(t_v).$$

The property we verify is that state *match* is never set:

$$\mathbf{G} \neg \mathit{match}.$$

In addition, we specify the environment such that it enforces the desired FDs on the chain. Concretely, we require that the values t_i, c_i returned for a requested s_i are the consistently the same throughout the run:

$$\begin{array}{l}
\forall s_u \forall t_u \forall c_u \forall s_v \forall t_v \forall c_v \forall t'_u \forall c'_u \forall s'_v \forall t'_v \forall c'_v\ \mathbf{G}(\\
(\mathbf{F}!\mathbf{chain}(s_u, t_u, c_u, s_v, t_v, c_v)) \wedge \\
(\mathbf{F}!\mathbf{chain}(s_u, t'_u, c'_u, s'_v, t'_v, c'_v)) \rightarrow t_u = t'_u \wedge c_u = c'_u)
\end{array}$$

and symmetrically for s_v. Notice that the environment specification is input-bounded (but non-strict). $\square$