Stellar: A Programming Model for Developing Protocol-Compliant Agents

Akın Günay and Amit K. Chopra

Lancaster University
Lancaster, LA1 4WA, UK
{a.gunay, amit.chopra}@lancaster.ac.uk

Abstract. An interaction protocol captures the rules of encounter in a multiagent system. Development of agents that comply with protocols is a central challenge of multiagent systems. Our contribution in this paper is a programming model, namely Stellar, to simplify development of compliant agents particularly for BSPL family of information-based interaction protocols. A significant distinction of Stellar from similar approaches is that it does not rely upon extracting control flow structures from protocol specifications to ensure compliance. Instead, Stellar provides a set of fundamental operations to programmers for producing viable messages according to the correct flow of information between agents, enabling flexible design and implementation of protocol-compliant agents. Our main contributions are: (1) identification of a set of programming errors that commonly occur when developing agents for protocol-based multiagent system, (2) definition of Stellar's operations and a simple yet effective pattern to develop protocol-compliant agents that avoid the identified errors, and (3) demonstration of Stellar's effectiveness by developing several agents in e-commerce and insurance policy domains.

1 Introduction

Interaction protocols capture the rules of encounter in multiagent systems by defining operational constraints on the occurrence and ordering of messages between agents. Effective interaction of agents in a multiagent system depends on their compliance with the system's protocol (i.e., the agents interact only according to the operational constraints that are defined by the protocol). However, development of protocol-compliant agents is challenging in many practical settings due to asynchrony of communication and the lack of infrastructure support for guaranteed message delivery while preserving their ordering.

There are several approaches to specify and implement interaction protocols, such as HAPN [14], Scribble [5, 15], BPMN in conjunction with BPEL [10], and business artifacts [6, 9]. These approaches mainly use procedural control flow structures (e.g., sequencing, branching, etc.) to specify interactions of agents, whose implementations reflect the protocol's control flow to ensure compliance. This is mostly achieved by developing agents on top of rigid code skeletons that are extracted from the protocol specifications. As a result, protocol specifications and implementations of agent who enact them become tightly coupled. An imminent drawback of this approach is the

lack of flexibility in agent design, which is a critical limitation particularly in open multiagent systems, where independent parties implement their own agents according to their private business requirements and logic. Another technical drawback of this approach is the need for synchronization between agents to ensure correct ordering of messages, which is hard to achieve in asynchronous decentralized environments.

Several information-based languages [11, 13, 3] have been proposed in the recent years to overcome limitations of the procedural protocol specification approaches. These languages specify protocols declaratively with respect to the correct flow of information between the agents. Hence, information-based languages do not impose a control flow for implementing protocol-compliant agents. Therefore, independent parties can design their agents as they see fit, as long as their agents emit messages follow the information flow as specified by the protocol. Consequently, information-based languages do not rely on synchronization and inherently support asynchronous communication.

In this paper we focus particularly on BSPL [11] which constitutes the base for all later information-based languages. Although BSPL provides a capable protocol specification language, it does not define a systematic methodology for developing protocol-compliant agents. Our contribution, namely Stellar, addresses this issue with a simple yet effective programming model. To this end, Stellar provides a programming pattern and a minimal set of fundamental operations to implement interaction of agents ensuring their compliance. The provided pattern and operations ensure compliance by separating an agent's interactions from its business logic. Thanks to BSPL's declarative approach, Stellar does not rely on control flow structures (e.g., no code skeleton is created), which enables maximum flexibility when designing and implementing agents. Our main contributions are as follows. One, we identify common pitfalls of protocol-compliant agent development in decentralized multiagent systems. Two, we develop Stellar's programming model, describe its programming pattern, and define its operations. Three, we demonstrate Stellar's effectiveness by developing several agents in e-commerce and insurance policy domains.

2 BSPL

BSPL [11] is an information-based protocol specification language. The main difference of BSPL from procedural protocol specification approaches is its way of characterizing operational constraints with respect to causality and flow of information between agents. We explain BSPL's main features using an example purchase protocol that we present in Listing 1.

Listing 1: A BSPL protocol for purchase.

```
Purchase {
 roles B, S // buyer, seller
 parameters out pID key, out item, out price, out result

 B ↦ S: rfq [out pID, out item]
 S ↦ B: quote [in pID, in item, out price]
 B ↦ S: accept [in pID, in item, in price, out result]
 B ↦ S: reject [in pID, in item, in price, out result]
```

```
}
```

A BSPL protocol is composed of a name, a set of roles, a set of public parameters, and a set of message references. BSPL is a declarative language and hence the ordering of message references is not important. The name of the protocol in Listing 1 is Purchase. It includes two roles, B and S corresponding to a buyer and seller, respectively. Purchase has four public parameters pID, item, price, and result, which describe the protocol's interface. A protocol's enactment is complete when all of its public parameters are bound. BSPL protocols can be composed using their interfaces to build complex interactions. However, we do not consider composite protocols in this paper for brevity. Each message reference $s \mapsto r : m[P]$ has a sender (s), a receiver (r), a name (m), and a set of parameters (P). For instance, the name of the first message reference in Listing 1 is rfq in which the sender is B, the receiver is S, and the parameters are pID and item. Instances of message references are defined over relational tuples that represent the bindings of message parameters. For instance, the following relation shows an instance of quote where pID, item, and price are bound to 1, $book$, and 5, respectively.

pID	item	price
1	$book$	5

In the rest of the paper we use "message" to refer both a message reference and message instance when there is no ambiguity. An *enactment* of a protocol is defined according to the set of messages that are exchanged between the agents. Each unique enactment of a protocol is identified by one or more public key parameters. In our example the only key parameter is pID. Hence, each distinct enactment of Purchase must have a unique binding for pID. When enacting a protocol, each agent keeps its own *local history*, which is the set of sent and received messages by the agent. Table 1 shows an example local history for the buyer, where there are four instances of Purchase (i.e., one for each distinct binding of pID). Note that only the enactments in which pID is bound to 1 and 2 are complete. That is, all the public parameters of Purchase are bound in these two enactments. The local history of an agent is sufficient for the agent to carry out its interaction with other agents. Hence, enactment of a BSPL protocol is fully decentralized and the knowledge of a global state is not needed.

Given an agent's local history, we say that a parameter's binding is *known* to the agent in an enactment of a protocol, if the agent's local history includes a message with a binding of the parameter for that particular enactment. Otherwise, we say that the parameter's binding is *unknown* to the agent in that particular enactment. For instance, according to the local history of the buyer in Table 1, binding of price is known (as 5) to the buyer for the enactment of Purchase where pID is bound to 1. This is due to the quote message that is received by the buyer for this enactment. However, the binding of price is unknown to the buyer for the enactment where pID is bound to 4, since there is no message in the buyer's local history with a binding of price for that enactment.

As we have stated earlier, the key idea of BSPL is to specify operational constraints of a protocol in terms of information flow, instead of using procedural control structures. BSPL models the flow of information in a protocol by adorning parameters with ⌜in⌝, ⌜out⌝, or ⌜nil⌝. Parameters that are adorned ⌜in⌝ in a message correspond conceptually to the inputs of the message, whose bindings must be known to the sender

pID	item
1	*book*
2	*bike*
3	*phone*
4	*pen*

(a) rfq

pID	item	price
1	*book*	5
2	*bike*	10
3	*phone*	20

(b) quote

pID	item	price	result
1	*book*	5	*OK*

(c) accept

pID	item	price	result
2	*bike*	10	*NOK*

(d) reject

Table 1: Local history of the buyer.

before sending the message. For instance, the seller must know the bindings of pID and item before sending a quote. Parameters adorned ⌜out⌝ correspond conceptually to the outputs of a message, whose bindings are produced by the sender when sending the message. For instance, the seller must produce the binding of price when sending a quote. Furthermore, if two or more message share the same ⌜out⌝ adorned parameter, only one of these messages can be sent in an enactment. Hence, the messages that share ⌜out⌝ adorned parameters are mutually exclusive (e.g., accept and reject. Lastly, if a parameter is adorned ⌜nil⌝ in a message, the sender must not know the binding of the parameter when sending the message. Agents cannot violate integrity of information when enacting a protocol. That is, a sender cannot change the known binding of a parameter when sending a message.

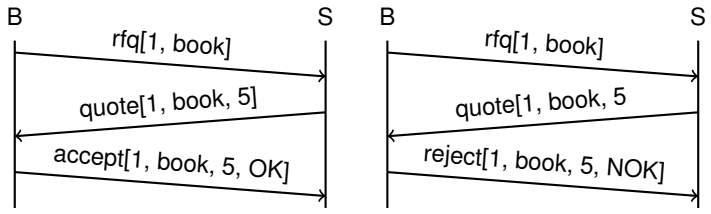
BSPL formalizes the correct flow of information in a protocol by defining *viability* of messages in an enactment. A message is viable for a sender in an enactment, if and only if (1) the sender knows the bindings of all the ⌜in⌝ adorned parameters of the message, and (2) there is no earlier message in the sender's local history that already binds one or more of ⌜out⌝ or ⌜nil⌝ adorned parameters of the message. Agents exchange only viable messages during a correct enactment of a protocol. For instance, considering the local history of the buyer in Table 1, the instance $(3, phone, 10, \mathit{OK})$ of accept is viable for the buyer, since the bindings of all its ⌜in⌝ adorned parameters are known (due to the received quote message from the seller) and the binding of the ⌜out⌝ adorned result is unknown to the buyer for the enactment where pID is 3. Hence, the buyer can send this message by producing the binding of result, which is *OK* in our case. Similarly, the reject message instance $(3, phone, 10, \mathit{NOK})$ is also viable.

On the other hand, there is no viable accept message for the enactment where pID is bound to 2, since the ⌜out⌝ adorned result is already bound to *NOK* in this enactment as a result of the prior reject message (i.e., the value of an ⌜out⌝ adorned parameter is known). Similarly, there is no viable reject message for the enactment where pID is bound to 1, since the ⌜out⌝ adorned result is already bound to *OK* in this enactment because of the prior accept message. For the enactment where pID is bound to 4, the buyer does not know the binding of price, which is adorned ⌜in⌝ in accept and reject messages. Hence, there is no viable accept or reject message for this enactment.

3 Pitfalls of Developing Compliant Agents

Programming of compliant agents for information-based protocols is a challenging task due to factors such as concurrent enactment of protocols and asynchronous communication between agents. Without a well-defined methodology, developers may easily fail to identify subtle details of protocols and implement non-compliant agents. In this section we identify such potential pitfalls of agent development for information-based protocols using our Purchase example from the previous section. Although our example is specified in BSPL, the issues that we discuss here are general and occur when developing agents for protocols that are specified in any information-based language.

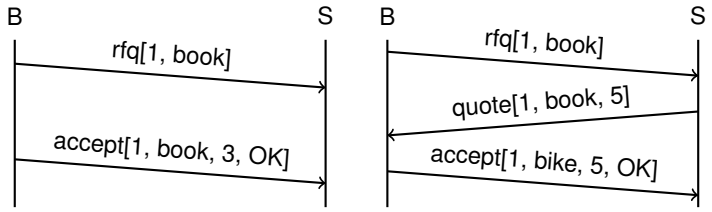
Let us start our discussion by examining some interactions between a compliant buyer and seller for our Purchase protocol. Figure 1 shows two such interactions. In both cases, the buyer first sends an rfq in which pID and item are bound to 1 and $book$, respectively. Then, the seller replies with a quote that binds price to 5. Finally, the buyer either sends an accept as in Figure 1(a) or a reject as in Figure 1(b) for the quote.



(a) Accept Quote (b) Reject Quote

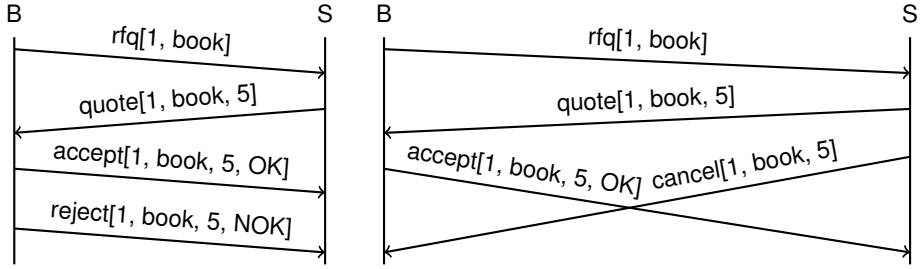
Fig. 1: Compliant interactions between the buyer (B) and seller (S).

Now we identify several issues that induce non-compliant implementation of agents.
Information Integrity: Protocol-compliant agents must ensure integrity of the exchanged information when interacting according to an information-based protocol. An agent may easily violate information integrity (maliciously or accidentally) by fabricating information that does not exist, or by altering known information. Figure 2(a) shows an interaction that corresponds to the former case, where the buyer sends an accept message to the seller without receiving a quote message by fabricating the binding of the price as 3. Figure 2(b) shows an interaction that corresponds to the latter case, where the buyer alters the binding of the item to $bike$ when sending the accept message, which should actually be $book$ as in the prior rfq message that she sent to the seller earlier. In both case, the integrity of the exchanged information is violated by the buyer, which result in non-compliant enactment of the protocol. These kind of mistakes occur especially when an agent is implemented for enacting multiple protocols at the same time. For instance, a developer may mistakenly use a parameter's binding in one protocol enactment as the binding of the same (or even different) parameter in another concurrent protocol enactment.
Mutual Exclusion: Realistic information-based protocols usually involve mutually exclusive messages. For instance, when the buyer receives a quote message from the seller,



(a) Value Fabrication (b) Value Alteration

Fig. 2: Violation of information integrity.



(a) Mutual Exclusion (Local) (b) Mutual Exclusion (Nonlocal)

Fig. 3: Violation of mutual exclusion.

she must either send an accept or reject message, but not both. Figure 3(a) shows violation of the mutual exclusion by the buyer, who sends first an accept message and then a reject message after receiving a quote message. Note that, in this example mutual exclusion is local to the buyer. That is, emission of the accept and reject messages are local choices of the buyer. Hence, violation of mutual exclusion can be avoided by ensuring the buyer's compliance with the protocol. However, mutual exclusion may also be non-local [7]. Suppose that in our purchase protocol the seller may send a cancel message between the quote and accept messages, to cancel its quote. Therefore, if there is a cancel message, there should not be an accept message and vice versa. However, these message are emitted by different agents (i.e., mutual exclusion is non-local) and violation of mutual exclusion may occur as Figure 3(b) shows. In general, non-local mutual exclusion cannot be achieved unless behaviors of agents are syncronized. However, in realistic multiagent systems achieving synchronization between agents is costly and hard (if not impossible).

Concurrency: As we have discussed in information integrity issues, in many practical multiagent systems, agents concurrently enact multiple protocols. For instance, in our purchase scenario, the buyer may concurrently send multiple quote requests to the seller for different items. Besides, the buyer (and also seller) can interact with multiple sellers (buyers) concurrently in different enactments of the purchase protocol. In such situations, in addition to the integrity issues that we have discussed, developers should also deal with interleaved asynchronous emission and reception of messages in different en-

actments. To this end, developers normally use multi-threading mechanisms (i.e., each concurrent protocol instance is executed in a separate thread). However, this requires the use of complex synchronization between the threads to properly handle interleaving messages from different enactments. Achieving such synchronization is an error-prone task and if not done correctly may easily cause agents to act in a non-compliant manner or event stop operating due to deadlock and liveness issues.

4 Stellar

Stellar[1] is a programming model that aims to simplify development of compliant agents for BSPL protocols by eliminating the issues that we have discussed in the previous section. To this end, Stellar provides a pattern via a set of well-defined operations to develop protocol-compliant agents. If an agent's interactions are implemented following Stellar's pattern, the developed agent is guaranteed to be protocol-compliant. We implement Stellar as a Java framework, in which agents are developed according to the following workflow. First, a BSPL protocol that represents the intended interactions of a set of agents is specified. Next, Stellar automatically generates a set of classes for each role in the protocol. These classes implement the fundamental operations to create and send viable messages according to Stellar's API. Finally, programmers develop their agents for each role using the provided set of classes. Before explaining Stellar's details, we first highlight Stellar's key features using the following Java snippet, which shows a possible implementation of the seller agent in Purchase to handle the reception of an rfq message and respond with the corresponding quote message.

```
1  public void handleRfq(Rfq rfq) {
2    Query query = new Query("pID", Query.EQ, rfq.get(Rfq.pID));
3    Quote quote = adapter.retrieveEnabled(Quote.class, query).getFirst();
4    // seller's business logic to determine the requested item's price
5    String price = priceMap.get(quote.get(Quote.ITEM));
6    quote.send(price);
7  }
```

Stellar follows BSPL's declarative approach. Hence, it does not impose a control flow for developing protocol-compliant agents. Instead, Stellar uses an event-driven model, where viable messages are created and sent according to the local history of an agent when certain events happen. In this regard, the above code snippet shows an event handler for the reception of an rfq message. The key class of Stellar is a role adapter. In our code snippet the seller's adapter is referred via the variable adapter, which is created during the initialization of the seller agent as we will demonstrate later. A fundamental feature of the adapter is to provide operations for retrieving *enabled* messages from an agent's local history. In an enabled message, all ⌜in⌝ adorned parameters are bound according to the local history of the agent, and all ⌜out⌝ and ⌜nil⌝ parameters are unbound. Hence, the programmer can easily create a viable message from an enabled message simply by producing bindings for all unbound ⌜out⌝ parameters. In this way Stellar ensures that agents send only viable messages and accordingly guarantees compliance of an agent's implementation with a protocol.

[1] Stellar is available on https://github.com/akingunay/stellar

To exemplify, in Line 3 of the above code snippet, adapter's retrieveEnabled method retrieves an enabled Quote message object to create a viable response to the received rfq message. The retrieval operation takes a query to determine which particular enabled message(s) it should retrieve. In our example, a single enabled Quote message object is retrieved as a response to the received rfq message, using the identifier of the received message in the query that is provided to retrieveEnabled. Finally, in Line 5, the send method of the retrieved quote message object is used to send the message to its recipient (i.e., the buyer who sent the received rfq message), which is automatically set by Stellar. Note that, in order to make the quote message object viable, the send method takes a price argument, whose value is determined by the seller's business logic. This value is used by Stellar to set the binding of the corresponding ⌜out⌝ adorned parameter.

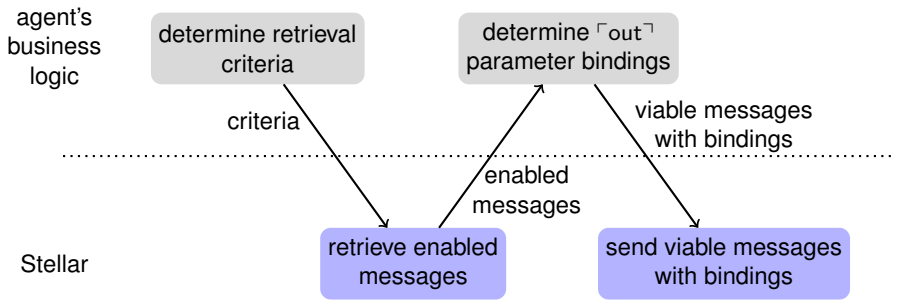


Fig. 4: Pattern to implement confirming agents using Stellar.

Figure 4 shows an abstract representation of Stellar's programming pattern that we use in the above code snippet to ensure compliance of the seller agent. First, the agent's business logic determines the criteria to retrieve a certain type of enabled message(s) from its local history. Then, the agent uses Stellar to retrieve the enabled message(s) that satisfy its criteria. Next, the agent's business logic determines the bindings of the ⌜out⌝ adorned parameters of the retrieved enabled message(s), and provide them to Stellar. Finally, Stellar compiles viable messages using the enabled messages and the provided bindings for the ⌜out⌝ adorned parameters, and sends them to their recipients.

4.1 Developing Agents using Stellar

In this section we present more details about Stellar using an example in which we implement a buyer agent for the Purchase protocol in Listing 1.

Structure of an Agent: Listing 2 shows the overall structure of the buyer's agent class Buyer. The object adapter of class BAdapter is the buyer's role adapter which is generated by Stellar from the specification of Purchase. Buyer class implements QuoteHandler interface, which is also generated by Stellar, with a single method handleQuote, which is called when the buyer receives a Quote message. The class may have other variables and methods as usual to represent the buyer's business logic.

Listing 2: Structure of the buyer's agent.

```
1  public class Buyer implements QuoteHandler {
2    private BAdapter adapter;
3
4    // class variables and methods to represent buyer's business logic
5    ...
6    // object initialization
7
8    public void handleQuote(Quote quote) {...}
9  }
```

Initialization of Agent: Listing 3 shows the constructor of Buyer class. The object adapter of class BAdapter is initialized using the factory method newAdapter according to a Configuration object, which includes information about the buyer's deployment (e.g., implementation details of its local history) and network addresses of the other agents. We discuss these concepts in detail in Section 4.2. Next, the created Buyer object registers itself as the handler for received Quote messages.

Listing 3: Initialization of the buyer's agent.

```
1  public Buyer(Configuration configuration) {
2    adapter = BAdapter.newAdapter(configuration);
3    adapter.registerQuoteHandler(this);
4    // initialization of other class variables
5    ...
6  }
```

Initialization of Interaction: Listing 4 shows how buyer agent initiates its interaction with a seller agent. The code first calls the start method of adapter. As a result, the buyer's agent becomes ready to send and receive messages. The second line retrieves an enabled rfq message object. Remember that in the Purchase protocol the rfq message does not have any ⌜in⌝ adorned parameters. Hence, this message corresponds to an entry point for a new enactment of the protocol. Therefore, the buyer can send this message at any time by setting its ⌜out⌝ adorned parameters. In other words, an rfq message is always viable. When retrieveEnabled method is called for such a message, Stellar automatically assigns values to the key parameter(s) of the message to create a unique key for initiating a new enactment ensuring information integrity. Hence, the only thing the buyer should do is to determine the item, for which it intends to request a quote, and call the send method of the retrieved rfq object. This code snippet can be part of any programmer defined method that captures the buyer agent's business logic.

Listing 4: Initialization of interaction by the buyer's agent.

```
1    adapter.start();
2    Rfq rfq = adapter.retrieveEnabled(Rfq.class);
3    String item = ... // set by buyer's business logic
4    rfq.send(item);
```

Handling of Messages: The next two code snippets show handling of received quotes. Listing 5 shows the interface that is generated by Stellar from the specification of Purchase and Listing 6 shows the implementation of the interface by the buyer's agent.

Listing 5: Specification of QuoteHandler interface.

```
1  public interface QuoteHandler {
2    public void handleQuote(Quote quote);
3  }
```

Remember that the buyer should either send an accept or reject message for a received quote to comply with Purchase. For simplicity, suppose that the business logic of the buyer is to accept quotes below 50 and reject others.

Listing 6: Implementation of QuoteHandler interface by buyer's agent.

```
1  public void handleQuote(Quote quote) {
2    Condition c1 = new Condition("pID", Query.EQ, quote.get(Quote.pID));
3    Condition c2 = new Condition("price", Query.LT, 50));
4    Query aQuery = new Query(new AndCondition(c1, c2));
5    Accept accept = adapter.retrieveEnabled(Accept.class, aQuery).getFirst();
6    if (accept != null) {
7      accept.send("OK");
8    } else {
9      Query rQuery = new Query("pID", Query.EQ, quote.get(Quote.pID));
10     Reject reject = adapter.retrieveEnabled(Reject.class, rQuery).getFirst();
11     reject.send("NOK");
12   }
13 }
```

The code in Listing 6 first creates a Query object to represent the buyer's acceptance criteria for the received quotes. The first part of the query calls the get method to determine the identifier of the enactment for which the quote is received (Line 2), and then defines the buyer's criterion for the acceptable value of the price (Line 3). Next, the code calls the retrieveEnabled method to retrieve an enabled Accept message object that matches to the given query. Note that there can only be one enabled accept message for every enactment with a particular binding of pID. However, retrieveEnabled returns a MessageSet object, which implements the Set interface with additional convenience methods. In Line 5, getFirst is one of these convenience methods that retrieves a single message if the set is a singleton and null otherwise. If there is an enabled accept message that matches the query (i.e., the quoted price is below 50), the code sends the retrieved Accept message using its send method, providing "OK" as the binding of result parameter (Line 7). Otherwise (i.e., the quoted price is above 50 and hence accept is null), the code sends a Reject message, which is retrieved by calling the retrieveEnabled method with the corresponding query.

4.2 Implementation of Stellar

Management of Local Histories: Stellar stores local history of an agent in a local relational database. Stellar hides the details of the particular database system from programmers. In fact, the programmer should not access the local history of the agent directly. Instead, the programmer should use only the provided retrieve and send methods by Stellar. Our implementation currently uses MySQL to store local histories of agents, however any relational database system that supports fundamental relational operations can be easily adopted.

Emission and Reception of Viable Messages: Stellar uses asynchronous message passing for agent communication, which is implemented using the UDP protocol. The messages are serialized into parameter-value pairs. Recent technologies such as JSON can easily be integrated into Stellar for improving interoperability. Emission of messages is enabled only via the send methods of the generated message classes. The aim of these methods is to ensure that agents send only viable messages by binding all of the necessary parameters. Otherwise, these methods throw exception. Reception of

messages and their insertion into an agent's local history is handled by Stellar. Hence, Stellar does not provide any method to programmers for manual message reception. Instead, the programmers should implement handlers of the messages that they want to react, which are automatically called by Stellar when a message is received. This simplifies programming of agents in asynchronous settings.

Retrieval of Enabled Messages: Here we provide Stellar's algorithm for the retrieval of enabled messages from an agent's local history. Below, we use, $\mathbb{P}$ for a BSPL protocol, p for individual parameters, P, Q, K for lists (or sets if their ordering is not important) of parameters. We use calligraphic capital letters for relations in the local history of an agent, and apply standard relational algebra operators Π for projection, σ for selection, $\bowtie$ for natural join, $⟗$ for full outer join, and $⟕$ for left outer join. We also use the utility methods `allParams`, `keyParams`, `inParams`, `nilParams`, and `outParams` with a relation and protocol argument to access the set of all, key, ⌜in⌝, ⌜nil⌝, and ⌜out⌝ adorned parameters of the relation, respectively.

Algorithm 1: retrieveEnabled($\mathcal{M}, \phi, \mathbb{P}$)

$P_{in} \leftarrow$ `inParams`$(\mathcal{M}, \mathbb{P})$ `// ⌜in⌝ adorned parameters of` $\mathcal{M}$
$P_{nil} \leftarrow$ `nilParams`$(\mathcal{M}, \mathbb{P})$ `// ⌜nil⌝ adorned parameters of` $\mathcal{M}$
$P_{out} \leftarrow$ `outParams`$(\mathcal{M}, \mathbb{P})$ `// ⌜out⌝ adorned parameters of` $\mathcal{M}$
$K \leftarrow$ `keyParams`$(\mathcal{M}, \mathbb{P})$ `// key parameters of` $\mathcal{M}$
$\mathcal{W}_I \leftarrow \emptyset$
if P_{in} *is* $\emptyset$ **then**
  return $\{()\}$
$\mathcal{W}_I \leftarrow \bigcup_{\mathcal{N} \in \mathbb{P}} \Pi_K(\mathcal{N})$
foreach $p \in P_{in}$ **do**
  $Q \leftarrow K \cup \{p\}$
  $\mathcal{W}_p \leftarrow$ `createRelation`(Q)
  foreach $\mathcal{N} \in \mathbb{P}$ *such that* $p \in$ `allParams`$(\mathcal{N}, \mathbb{P})$ **do**
    $\mathcal{W}_p \leftarrow \Pi_Q(\mathcal{N}) \cup \mathcal{W}_p$
  $\mathcal{W}_I \leftarrow \mathcal{W}_I \bowtie_K \mathcal{W}_p$
$\mathcal{W}_E \leftarrow \bigcup_{\mathcal{N} \in \mathbb{P}} \Pi_K(\mathcal{N})$
foreach $p \in P_{out} \cup P_{nil}$ **do**
  $Q \leftarrow K \cup \{p\}$
  $\mathcal{W}_p \leftarrow$ `createRelation`(Q)
  foreach $\mathcal{N} \in \mathbb{P}$ *such that* $p \in$ `allParams`$(\mathcal{N}, \mathbb{P})$ **do**
    $\mathcal{W}_p \leftarrow \Pi_Q(\mathcal{N}) \cup \mathcal{W}_p$
  $\mathcal{W}_E \leftarrow \mathcal{W}_E ⟕_K \mathcal{W}_p$
$\mathcal{W} \leftarrow \sigma_{P_{out}=null \wedge P_{nil}=null}(\mathcal{W}_I ⟕_K \mathcal{W}_E)$
return $\sigma_\phi(\mathcal{W})$

Algorithm 1 defines retrieval of enabled messages, given the relation $\mathcal{M}$ that corresponds to a message reference (e.g., a quote message reference), the user defined query ϕ, and the protocol specification $\mathbb{P}$. Note that the algorithm returns a relation (not actual

message objects in Java), where each tuple of the relation corresponds to the parameter bindings of an enabled instance of the message reference $\mathcal{M}$. A Stellar adapter uses this relation to create the corresponding message objects and return them as the result of a retrieveEnabled method call as we demonstrated in earlier examples.

Algorithm 1 can be divided into three phases. In the first phase (from lines 8 to 14), the algorithm builds the relation $\mathcal{W}_I$ where all $\ulcorner\mathsf{in}\urcorner$ adorned parameters of $\mathcal{M}$ are bound. In the second phase (from lines 15 to 21), the algorithm builds another relation $\mathcal{W}_E$ where one or more $\ulcorner\mathsf{out}\urcorner$ or $\ulcorner\mathsf{nil}\urcorner$ adorned parameters of $\mathcal{M}$ are bound. In the last phase (lines 22–23), the algorithm removes the tuples from $\mathcal{W}_I$ for which there is a matching tuple (i.e., identified by the same key) in $\mathcal{W}_E$. Hence, each tuple of the resulting relation corresponds to an enabled message (i.e., all $\ulcorner\mathsf{in}\urcorner$ parameters are bound and all $\ulcorner\mathsf{out}\urcorner$ and $\ulcorner\mathsf{nil}\urcorner$ parameters are unbound). The algorithm applies the given query ϕ to the resulting relation $\mathcal{W}$ to filter the tuples.

4.3 Revisiting Pitfalls

Stellar's retrieval and emission operations ensure causality and information integrity. Specifically, retrieval operations ensure integrity of $\ulcorner\mathsf{in}\urcorner$ adorned parameters by binding them permanently to the corresponding values according to the agent's local history. Hence, a programmer cannot fabricate or alter $\ulcorner\mathsf{in}\urcorner$ adorned parameters of a message. The send operations ensure integrity of $\ulcorner\mathsf{out}\urcorner$ and $\ulcorner\mathsf{nil}\urcorner$ adorned parameters by enforcing the programmer to assign values to only $\ulcorner\mathsf{out}\urcorner$ adorned parameters when needed. Hence, the programmer cannot omit assignment of mandatory parameters and thus break integrity. Further, Stellar handles creation of bindings for key parameters ensuring their uniqueness, which prevents key related integrity issues.

Stellar's retrieval operations prevent emission of mutually exclusive messages. That is, if two (or more) messages are mutually exclusive in a protocol and an agent has already sent one of these messages in an enactment of the protocol, the retrieval operation does not consider the other mutually exclusive message(s) as enabled in the same enactment. Hence, the agent cannot retrieve and send mutually exclusive messages. Stellar does not directly handle non-local mutual exclusion. However, safety of a BSPL protocol, which means that the protocol is free from non-local mutual exclusion, can be verified automatically [12] at design time to avoid non-local mutual exclusion issues.

Communication in Stellar is fully asynchronous. Hence a single-threaded agent can easily enact multiple protocols at the same time using Stellar. That is, an agent's execution is never blocked when sending or receiving messages. Further, Stellar's programming model handles one incoming message at a time. Hence, developers can implement their agents without any thread synchronization that deals with interleaving reception of multiple messages in different enactments. This feature of Stellar substantially simplifies design of an agent, as our case study in Section 5 demonstrates. Note that UDP, which is used in our implementation, is an unreliable protocol (i.e., it does not guarantee delivery of emitted messages), which may compromise liveness of an interaction. This issue can be avoided using a reliable alternative, such as RUDP, TCP, or message queues. However, these alternatives provide features (e.g., ordered message delivery), which are not needed by Stellar. We chose UDP for our implementation to show that

lack of such features do not affect protocol-compliance of agents. We will address liveness of interactions in our future work.

5 Case Study

To demonstrate the use of Stellar in a more comprehensive case, where an agent should consider multiple messages for decision making, we use a claim handling scenario from insurance domain. We list the protocol that represent this scenario in Listing 7. In this scenario there is a policy subscriber and an insurer. The subscriber can make multiple claims (claim message) by sending an incident's details and the claimed amount to the insurer. The insurer either approves (approve message) or rejects a claim (reject message). In case of approval, the insurer pays the claimed amount to the subscriber. The insurer can pay its balance immediately for each claim or as lump sum for several claims (pay message). For brevity, we omit some policy aspects such as premium payments.

Listing 7: An insurance policy claim protocol.

```
Insurance {
 roles I, S //insurer, subscriber
 parameters out sID key, out cID key, out pID key, out subscriber, out period,
      out type, out date, out incident, out cAmount, out outcome, out pAmount

 S ↦ I: subscribe[out sID, out subscriber, out period, out type]
 I ↦ S: register[in sID, in subscriber, in period, in type, out date]
 S ↦ I: claim[in sID, out cID, out incident, out cAmount]
 I ↦ S: approve[in sID, in cID, in incident, in cAmount, out outcome]
 I ↦ S: reject[in sID, in cID, in incident, in cAmount, out outcome]
 I ↦ S: pay[in sID, out pID, out pAmount]
}
```

Listing 8 shows the implementation of ClaimHandler interface by the insurer agent to handle claim messages when enacting Insurance. As we explained earlier, ClaimHandler interface is generated by Stellar from the specification of Insurance and consists of a single method handleClaim, which is used to define the insurer's business logic for handling claims. Suppose that the insurer handles a claim in two steps. In the first step, the insurer decides whether the received claim is valid or not. In the second step, if the claim is valid and the insurer's policy balance exceeds a minimum payable amount, the insurer pays its balance. Otherwise, the insurer does not make any immediate payment.

The method processClaim (Lines 8–21) captures the first step. The method first decides whether the received claim is valid by calling isValidClaim (Line 12), which returns true for valid and false for invalid claims. We do not present the details of isValidClaim since they are not relevant to our demonstration. Depending on the validity of the claim, processClaim retrieves and sends either the enabled Approve (Lines 13–15) or Reject (Lines 17–19) message. Finally, processClaim returns true or false depending on the validity of the claim.

If the claim is approved (Line 3), handleClaim calls payBalance (Line 4), which captures the second step (Lines 23–30). The method payBalance first computes the insurer's total balance for the policy using approvedClaimAmount and paidClaimAmount methods (Line 24). We describe these methods later in Listing 9. If the insurer's balance is more that the minimum payable amount, processClaim retrieves and sends the enabled Pay message to pay the insurer's balance (Lines 25–29).

Listing 8: Implementation of ClaimHandler interface by the insure agent.

```
1  public void handleClaim(Claim claim) {
2    boolean isApproved = processClaim(claim);
3    if (isApproved) {
4      payBalance(claim.get(Claim.sID));
5    }
6  }
7
8  private void processClaim(Claim claim) {
9    Condition c1 = new Condition("sID", Query.EQ, claim.get(Claim.sID));
10   Condition c2 = new Condition("cID", Query.EQ, quote.get(Claim.sID));
11   Query query = new Query(new AndCondition(c1, c2));
12   if (isValidClaim(claim)) {
13     Approve msg = adapter.retrieveEnabled(Accept.class, query).getFirst();
14     msg.send("APPROVED");
15     return true;
16   } else {
17     Reject msg = adapter.retrieveEnabled(Reject.class, query).getFirst();
18     msg.send("REJECTED");
19     return false;
20   }
21 }
22
23 private void payBalance(String sId) {
24   int balance = payableClaimedAmount(sId) - totalPaidAmount(sId);
25   if(MIN_PAYABLE_AMOUNT <= balance) {
26     Query query = new Query("sID", Query.EQ, sId);
27     Pay msg = adapter.retrieveEnabled(Pay.class, query);
28     msg.send(balance);
29   }
30 }
```

Computation of the insurer's balance for a policy requires consideration of multiple messages. That is, we should first compute the total payable claimed amount for the policy according to the approved claims. Then we should compute the total paid amount for the policy according to the previous payments, and subtract it from the total payable claimed amount. The method payableClaimedAmount in Listing 9 (Lines 1–9) computes the total payable claimed amount. It first retrieves all the sent Approve messages for the policy, which is identified by sId, from the insurer's message history calling retrieveMessage method (Line 3). Note that this is a different method than retrieveEnabled. Next, the total payable amount is computed by iterating over all the retrieved Approve messages and summing up the claimed amount of each message. The method totalPaidAmount (Lines 11–19) repeats the same process to compute the total paid amount for the policy using Pay messages (instead of Approve messages) and corresponding paid amounts in those messages.

Listing 9: Computation of total claimed and paid amounts.

```
1  private int payableClaimedAmount(String sId) {
2    Query query = new Query("sID", Query.EQ, sId);
3    MessageSet<Approve> msgs = adapter.retrieveMessage(Approve.class, query);
4    int sum = 0;
5    for (Approve msg : msgs) {
6      sum += (int) msg.get(Approve.cAmount);
7    }
8    return sum;
9  }
10
11 private int totalPaidAmount(String sId){
12   Query query = new Query("sID", Query.EQ, sId);
13   MessageSet<Pay> msgs = adapter.retrieveMessage(Pay.class, query);
14   int sum = 0;
```

```
15      for (Pay msg : msgs) {
16        sum += (int) msg.get(Pay.pAmount);
17      }
18      return sum;
19    }
```

6 Discussion

This paper presented Stellar, a programming model for developing compliant agents for BSPL protocols. Stellar's main idea is to ensure compliance of agents by allowing exchange of only viable messages between them. To this end, Stellar provides a simple yet effective programming model for retrieving enabled messages from an agent's local history and for sending them ensuring their viability. Communication in Stellar is fully asynchronous. Further, Stellar implicitly ensures information integrity of interaction, prevents emission of mutually exclusive messages, and enables concurrent enactment of protocols without relying on thread synchronization. Accordingly, Stellar simplifies agent development in decentralized settings ensuring their compliance. Stellar is different from programming models that require agent compliance with control flows, and more distantly related to distributed programming models without interaction protocols.

Scribble [15] enforces design-time adherence to protocols that specify typed message signatures and messaging constraints with explicit control flow. Scribble [5] extracts state machines from protocol specifications to generate APIs for endpoints. Sending or receiving a message returns a protocol state object and each protocol state object provides message sending and receiving operations that comply with correct subsequent state transitions. Stellar provides more flexibility to developers, since it does not impose a control flow for ensuring compliance when implementing agents. Developers are free to design their agents as they see fit, without concerning about the state of their interactions. Stellar's retrieve and send pattern ensures exchange of only viable messages and accordingly ensures compliance of agent, which is independent from a control flow.

Business-oriented approaches for web services propose the use of high-level processes according to which correct interactions are enforced in code. Business Process Modeling Notation (BPMN) has been used to specify processes that are then translated into the Business Process Execution Language [10] (BPEL), an executable language for externally invoking web services and their interactions based on event occurrences. The BPMN-BPEL approach is inherently process-oriented and depends on the correct realization of workflows. Stellar is information-oriented and uses a declarative approach without imposing any workflow. Business artifacts [9] are high-level representations of both processes, interaction, and the relational data that they operate on. Artifact interoperation hubs [6] enforce correct messaging with business processes by acting as central communication points between web services. Stellar only uses local information and does not rely on any centralized communication artifacts to ensure compliance.

Further afield, are programming models for distributed systems that do not consider protocols. For example, functional reactive programming [2], the Sunny Event-driven programming model [8], and the Actor programming model [4] implemented in Akka [1]. These programming models support interaction derived from internal system or actor code, without a protocol specification against which correct implementations must

comply; we focus on supporting independent agent development against protocol specifications, where programmers are protected from violating protocol conformance and integrity of information. We will investigate how Stellar's ideas can be integrated into these programming models in our future work.

Acknowledgements: We would like to thank Munindar P. Singh and Thomas C. King for their valuable comments to improve this paper. Akın Günay and Amit K. Chopra were supported by the EPSRC grant EP/N027965/1 (Turtles).

References

1. Akka: 2.5.6 (2017), http://akka.io
2. Bainomugisha, E., Carreton, A.L., van Cutsem, T., Mostinckx, S., de Meuter, W.: A survey on reactive programming. ACM Computing Surveys **45**(4), 52:1–52:34 (2013)
3. Chopra, A.K., Christie V., S.H., Singh, M.P.: Splee: A declarative information-based language for multiagent interaction protocols. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 1054–1063 (2017)
4. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. pp. 235–245. IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)
5. Hu, R., Yoshida, N.: Hybrid session verification through endpoint api generation. In: Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering - Volume 9633. pp. 401–418. Springer-Verlag, New York, NY, USA (2016)
6. Hull, R., Narendra, N.C., Nigam, A.: Facilitating workflow interoperation using artifact-centric hubs. In: Proceedings of the 7th International Joint Conference on Service-Oriented Computing. pp. 1–18. ICSOC-ServiceWave '09, Springer-Verlag, Berlin, Heidelberg (2009)
7. Ladkin, P.B., Leue, S.: Interpreting message flow graphs. Formal Aspects of Computing **7**(5), 473–509 (1995)
8. Milicevic, A., Jackson, D., Gligoric, M., Marinov, D.: Model-based, event-driven programming paradigm for interactive web applications. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 17–36. Onward! 2013, ACM, New York, NY, USA (2013)
9. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal **42**(3), 428–445 (2003)
10. Ouyang, C., Dumas, M., van der Aalst, W.M.P., Hofstede, t.A.H.M., Mendling, J.: From business process models to process-oriented software systems. ACM Transactions on Software Engineering and Methodology **19**(1), 2:1–2:37 (2009)
11. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems. pp. 491–498 (2011)
12. Singh, M.P.: Semantics and verification of information-based protocols. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems. pp. 1149–1156 (2012)
13. Singh, M.P.: Bliss: Specifying declarative service protocols. In: Proceedings of the 2014 IEEE International Conference on Services Computing. pp. 235–242 (2014)
14. Winikoff, M., Yadav, N., Padgham, L.: A new hierarchical agent protocol notation. Autonomous Agents and Multi-Agent Systems **32**(1), 59–133 (2018)
15. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: 8th International Symposium on Trustworthy Global Computing - Volume 8358. pp. 22–41. TGC 2013, Springer-Verlag New York, Inc., New York, NY, USA (2014)