Rafael H. Bordini, Louise A. Dennis and Yves Lespérance (eds)
Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019)
13th-14th May 2019, Montreal, Canada
cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_28.pdf

Concurrency and Asynchrony in Protocol Languages

Amit K. Chopra[1], Samuel H. Christie V[2], and Munindar P. Singh[3]

[1] Lancaster University amit.chopra@lancaster.ac.uk
[2] Lancaster University s.christie@lancaster.ac.uk
[3] North Carolina State University singh@ncsu.edu

Abstract. Despite shared objectives, modern languages for specifying multiagent interaction protocols differ significantly in their—often complex—technical details. We contribute a comparative evaluation of prominent select languages based on their support for concurrent enactments and for asynchrony, both important facets of flexibility. We show how the underlying abstractions and assumptions of the various languages fare on these criteria.

1 Introduction

Protocols lie at the heart of multiagent systems (MAS). Specification languages for protocols are therefore crucial in engineering MAS. Several notable MAS software methodologies, e.g., [11,4,12], give a place of prominence to specifying protocols; however, most rely on informal notations such as AUML [10] for specifying them.

Over the last two decades, several formal languages for specifying protocols have been proposed, and in communities as diverse as programming languages, Web services, and multiagent systems. These languages make different assumptions and provide varying capabilities for specifying protocols. However, today, we lack generally clear evaluation criteria or use cases for protocol languages. Any set of evaluation criteria would necessarily be incomplete, but what would constitute an informative set that sheds light upon the theoretical foundations for protocols? To answer this question, we turn to some central challenges in specifying interaction protocols.

A central challenge in specifying interaction protocols for MAS is how to enable the specification of *flexible* protocols that agents can *correctly* enact in a decentralized manner, that is, based solely only on local knowledge and accommodating asynchrony? This challenge captures the essence of what drives most theoretical work on protocol languages [3,7,2,15,5,9]. The difficulty is that flexibility, which includes important aspects such as concurrency, is in tension with correctness in asynchronous settings.

Based on the foregoing, we motivate the following criteria for evaluating protocol languages. How well does a language support *concurrent* enactments? How well does a language support *asynchrony*? For example, does it require *ordered delivery* of messages?

Using these criteria, we undertake a comparative evaluation of selected approaches. We focus on state-of-the-art languages. Specifically, we evaluate each approach with respect to the same criteria by specifying (as best possible) the same scenarios.

The paper's overarching *contribution* and *significance* lie in developing and applying unified evaluation criteria for protocol languages. Its *novelty* arises from the absence, currently, of such a framework. Notably, this paper focuses on essential representational criteria and plays down contingent features such as current tool support and popularity.

2 Overview of Selected Approaches

We select protocol specification languages that are recent and represent diverse doctrines. We introduce their main ideas via a common scenario in which a buyer B requests an item from a seller S, who responds with a price. B may accept or reject the offer. Rejection ends the enactment. If B accepts, S instructs warehouse W to ship the item, following which W delivers it to B.

.

2.1 Multiparty Session Types

We discuss two prominent approaches, Trace and Scribble.

Trace Expressions. Castagna et al. [5], Ancona et al. [1], and Ferrando et al. [8] exemplify this approach. Hereon, we refer to this approach as Trace. We follow Castagna et al.'s [5] variant for concreteness as they give clear rules for determining projections of protocols. Here, $x \xrightarrow{m} y$ means that x sends message m to y; ';' denotes sequence, '$\vee$' denotes mutually exclusive choice, and '$\wedge$' denotes shuffle (order-preserving interleaving). Trace assumes pairwise-FIFO communication. Listing 1 shows how our purchase example may be rendered in Trace.

Listing 1: Three-party *Purchase* protocol in Trace.

```
B --Item--> S ; S --Price--> B ;
(B --Accept--> S ; S --Ship--> W ; W --Deliver--> B) ∨ B --Reject--> S
```

Given a protocol, Trace yields *projections* for each role. A role's projection represents the local protocol-related computations performed by the role. Ideally, the computations realized jointly by all projections of a protocol should exactly be the computations of the protocol (as we shall see, this is not always the case).

Scribble. Scribble [19] (which is itself based on [9]) is similar to Trace. A protocol is an ordering of constituent protocols (bottoming out at individual message specifications) using constructs such as sequence, choice, and iteration. Scribble assumes ordered communication over FIFO channels. Listing 2 highlights its salient features.

Listing 2: Three-party *Purchase* in Scribble.

```
protocol Purchase(role B, role S, role W) {
 Item(string) from B to S;
 Price(int) from S to B;

 choice at B {
   Accept() from B to S;
   Ship() from S to W;
   Deliver() from W to B;
  } or {
   Reject() from B to S;
   NoShip() from S to W; }}
```

Scribble [14] has tooling that can generate role-specific projections from protocols.

2.2 HAPN

HAPN [17] is a graphical protocol language. As Figure 1 shows, nodes represent states; they can also reference other protocols to compose them. Edges can have complex annotations, supporting the specification of message transmissions, guard expressions, and changes to state. HAPN specifies the computations of a protocol in terms of state machines. It assumes synchronous communication [17, p. 61] and does not give a method for projecting a protocol to local perspectives (although it acknowledges their need).

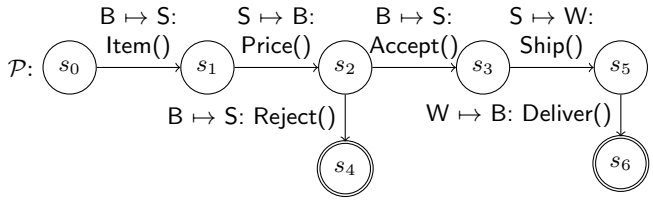


Fig. 1: Three-party *Purchase* in HAPN, starting from s_0.

HAPN provides methods to flatten a hierarchical protocol into simple protocols and finite state machines for verification.

2.3 BSPL

BSPL [15,16] and Splee [6], which extends BSPL, are exemplars of information-based languages, which are declarative and eschew specifying ordering between messages. In BSPL, a protocol specifies an information object. Each protocol enactment fills out an instance of this object. A protocol specifies two kinds of constraints on the messages a role can observe: *causality* or information flow constraints between protocols and *integrity* or consistency constraints on any object.

Listing 3: Three-party *Purchase* in BSPL.

```
Purchase {
 role B, S, W
 parameter out ID key, out item, out price, out d, out OK

 B ↦ S: RFQ[out ID, out item]
 S ↦ B: Offer[in ID, in item, out price]
 B ↦ S: Accept[in ID,in item,in price,out d,out addr]
 B ↦ S: Reject[in ID, in price, out d, out OK]
 S ↦ W: Ship[in ID, in item, in addr]
 W ↦ B: Deliver[in ID, in item, in addr, out OK] }
```

In Listing 3, B requests a quote from S for a specific item, and initiates an enactment uniquely identified by key ID. The ⌜out⌝ annotations on ID and item mean that in sending *RFQ*, B produces bindings for those parameters. The ⌜in⌝ annotations on ID and item in the *Offer* message indicate that S needs to *know* these parameters before S can send *Offer*. B can either accept or reject the offered price, but not both, because both messages produce a "decision" (as d), and integrity requires a parameter to have at most one binding. An accept message includes the buyer's address, so S can send shipping instructions to warehouse W. An enactment is complete when all public ⌜out⌝ parameters are bound, so either *Reject* or *Deliver* can complete *Purchase*. Notice that S cannot send *Ship* if *Reject* happens as it does not know addr.

The projection of a BSPL protocol to a role is a protocol containing only those messages that involve the specified role. For example, the projection of *Purchase* for W consists only of *Ship* and *Deliver*.

3 Concurrent Enactments

To support autonomy, we should constrain an agent only as essential to the enactment of a protocol. In particular, we should allow agents to act concurrently when doing so would not violate correctness.

Scenario. B sends *Request* to S to ship some item. After sending *Request*, B may send *Payment*. After receiving *Request*, *S* may send *Shipment*. That is, *Payment* and *Shipment* are not mutually ordered.

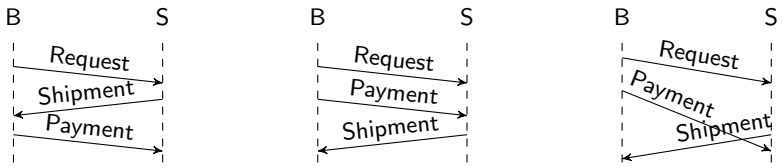
This scenario is natural in case of reciprocal commitments: B commits to S that if *Shipment* occurs, B will send a *Payment* and S commits to B that if *Payment* occurs, S will send a *Shipment* [18].

Figure 2 shows possible enactments of one protocol instance, eliding the parameters.

Listing 4 gives a Trace protocol specification that appears to capture the scenario.

Listing 4: Attempt to capture Figure 2 in Trace.

$$\mathsf{B} \xrightarrow{\mathsf{Request}} \mathsf{S};\ (\mathsf{B} \xrightarrow{\mathsf{Payment}} \mathsf{S} \wedge \mathsf{S} \xrightarrow{\mathsf{Shipment}} \mathsf{B})$$



(a) Shipment first. (b) Payment first. (c) Concurrent.

Fig. 2: Three possible enactments of *Purchase*.

The shortcomings of Listing 4 become apparent when determining its projections. Following Trace [5, p. 14], we eliminate $\wedge$ from Listing 4 to obtain the equivalent Listing 5.

Listing 5: A Trace transformation of protocol in Listing 4.

$$\mathsf{B} \xrightarrow{\mathsf{Request}} \mathsf{S};\ (\mathsf{B} \xrightarrow{\mathsf{Payment}} \mathsf{S}\ ;\ \mathsf{S} \xrightarrow{\mathsf{Shipment}} \mathsf{B}) \vee (\mathsf{S} \xrightarrow{\mathsf{Shipment}} \mathsf{B};\ \mathsf{B} \xrightarrow{\mathsf{Payment}} \mathsf{S})$$

Listing 5 is unprojectable because the choice is controlled by different parties—either B pays or S ships. Specifically, the projections must interpret $\vee$ as external choice (choosing) for one agent and an internal choice (following) for another. Listing 6 arbitrarily gives the external choice (denoted $+$) to B and the internal choice (denoted $\oplus$) to S (the reverse is equally good since the situation is symmetric). However, this provides only the illusion of choice, because choosing to receive causes deadlock; the agent arbitrarily given the internal choice *must* send first, enabling only one of the two desired enactments.

Listing 6: Hypothetical local projections in Trace that illustrate the difficulty with choice.

$$\mathsf{B}:\ \mathsf{S}!\mathsf{Request}.(\mathsf{S}?\mathsf{Shipment}.\mathsf{S}!\mathsf{Payment}) + (\mathsf{S}!\mathsf{Payment}.\mathsf{S}?\mathsf{Shipment})$$
$$\mathsf{S}:\ \mathsf{B}?\mathsf{Request}.(\mathsf{B}!\mathsf{Shipment}.\mathsf{B}?\mathsf{Payment}) \oplus (\mathsf{B}?\mathsf{Payment}.\mathsf{B}!\mathsf{Shipment})$$

Castagna et al. formalize properties of *sequentiality* and *knowledge for choice* under which Listing 6 is invalid because of the nonlocal choice between *Payment* and *Shipment*.

Scribble shares Trace's limitations. Listing 7 shows how we might model the scenario in Scribble. Scribble tooling rejects the protocol as ill-formed because the choice is at B but one of the alternatives triggers with an action by S.

Listing 7: Encode in Scribble.

```
protocol FlexiblePurchase (role B, role S) {
  Request() from B to S;
  choice at B {
    Payment() from B to S;
    Shipment() from S to B;
  } or {
    Shipment() from S to B; // not valid
    Payment() from B to S;}}
```

Figure 3's HAPN protocol captures only the first two enactments, not the concurrent one, because HAPN assumes synchrony.

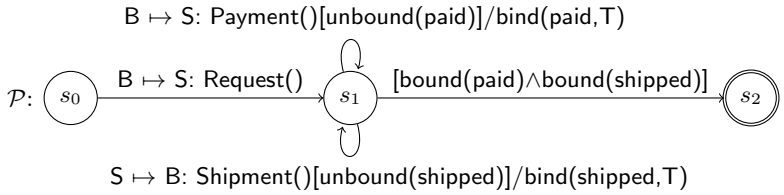


Fig. 3: *FlexiblePurchase* in HAPN.

Listing 8 gives a BSPL protocol. It supports the enactment in Figure 2c because after B sends *Request*, it has the information needed to send *Payment* and, upon receiving *Request*, S has the information needed to send *Shipment*.

Listing 8: A protocol exhibiting concurrency.

```
Flexible Purchase {
 role B, S
 parameter out ID key, out item, out shipped, out paid

 B ↦ S: Request[out ID, out item]
 S ↦ B: Shipment[in ID, in item, out shipped]
 B ↦ S: Payment[in ID, in item, out paid] }
```

4 Asynchrony

Whereas synchronous communication couples a sender and receiver (they must be ready to receive and send at the same time), asynchronous communication does not. Asynchrony is supported by the Internet and promotes decentralization: agents do not need to know of each other's states or wait for each other. Scribble, Trace, and BSPL support asynchrony; HAPN [17, p. 61] does not. However, digging deeper, we uncover significant differences between Scribble and Trace on the one hand and BSPL on the other. The differences stem from the fact that Scribble and Trace both require pairwise-FIFO delivery of messages whereas BSPL does not.

A consequence of the end-to-end principle [13] is that a protocol should not rely on message ordering guarantees from the communication infrastructure since the appropriate constraints are to be checked in an upper layer. Relying on such guarantees naturally limits the kinds of communication infrastructures upon which a protocol may be used. Further, protocols for lightweight communications (e.g., for IoT) or fast interactions (e.g., in financial transactions) cannot support FIFO. And, FIFO is inadequate for settings of more than two parties as the following scenario demonstrates.

Scenario. In an indirect-payment purchase protocol, after receiving an *Offer* B sends *Accept* to S and then *Instruct* (a payment instruction) to bank K. Upon receiving *Instruct*, K does a funds *Transfer* to S.

(a) In-order delivery. (b) Out-of-order delivery.

Fig. 4: FIFO does not guarantee stable ordering.

Figure 4 shows two enactments for this scenario. In Figure 4a, S receives *Accept* before *Transfer* whereas in Figure 4b, S receives it after *Transfer*. Both enactments satisfy pairwise FIFO. The enactments illustrate that even with FIFO ordering, asynchrony makes ordering indeterminate for protocols involving more than two agents. Listing 9 gives a protocol that supports both enactments in Figure 4.

Listing 9: Indirect payment protocol in BSPL.

```
Indirect Payment {
 role B, S, K // K is bank
 parameter out ID key, out item, out price, out acc, out inst,
     out OK

 S ↦ B: Offer[out ID, out item, out price]
 B ↦ S: Accept[in ID, in item, in price, out acc]
 B ↦ K: Instruct[in ID, in price, in acc, out inst]
 K ↦ S: Transfer[in ID, in price, in inst, out OK] }
```

Listing 10 is an attempt to capture the scenario in Trace. Following Trace [5, p. 16], this protocol is not well-formed, which means that the correctness of its projections, also given in Listing 10, cannot be guaranteed. Specifically, the projection for S expects *Accept* before *Transfer* and therefore does not support the enactment in Figure 4b, which may arise despite using FIFO channels.

Listing 10: A plausible indirect payment protocol and its projections in Trace.

```
//Protocol
S -Offer-> B; B -Accept-> S; B -Instruct-> K; K -Transfer-> S
//Projections
B: S?Offer.S!Accept.K!Instruct
S: B!Offer.B?Accept.K?Transfer
K: B?Instruct.S!Transfer
```

Listing 11 gives a Scribble protocol to capture the scenario. The projections given are produced by Scribble tooling, which verifies the specification. The projections are analogous to the Trace projections in Listing 10; in particular, S cannot receive *Transfer* before *Accept*; it blocks on the reception of Accept on the channel from B even if *Transfer* may have arrive earlier on the channel from

K. Effectively, the projection reorders the receptions of the two messages. The listing shows S's projection (other roles' projections are elided).

Listing 11: Indirect payment in Scribble.

```
protocol IndirectPayment(role S, role C, role B) {
 Offer() from S to B;
 Accept() from B to S;
 Instruct() from B to K;
 Transfer() from K to S; }

projection IndirectPayment_S(role B, role S, role K) {
 Offer() to B;
 Accept() from B;
 Transfer() from K; }
```

Reordering messages as Scribble does undesirable for many reasons. One, processing of messages that have arrived earlier is unnecessarily delayed. Two, blocking paves the way for deadlocks. For example, imagine a scenario where an agent is waiting for m_1 to arrive, with m_2 already in its buffer waiting to be received by the agent. If m_2 disables the emission of m_1, then the agent is deadlocked. What the foregoing scenario shows is that despite the FIFO assumption, both Scribble and Trace (unnecessarily) rule out realistic message orders that are simply the result of asynchrony.

5 Summary

Table 1: Summary of evaluation.

Criterion	Scribble	Trace	HAPN	BSPL
Concurrency	No	No	No	Yes
Asynchrony	Yes	Yes	No	Yes
Unordering	No	No	–	Yes

Table 1 summarizes our findings. Our criteria and scenarios are elementary, motivated from fundamental challenges for interaction protocols. And our selected approaches represent recent research into protocols. Our evaluation is concrete and comparative, driven by the specification of scenarios in the selected approaches, followed by their analysis. Our evaluation shows significant advantages of BSPL over the other approaches. The results presented here can be a starting point for a more extensive comparison of protocol languages.

Acknowledgments. We thank the reviewers for providing useful comments. Chopra and Christie were supported by EPSRC grant EP/N027965/1 (*Turtles*). Singh thanks the US Department of Defense for partial support under the Science of Security Lablet.

References

1. Ancona, D., Ferrando, A., Franceschini, L., Mascardi, V.: Coping with bad agent interaction protocols when monitoring partially observable multiagent systems. In: Proceedings of the 16th International Conference on Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity (PAAMS). Lecture Notes in Computer Science, vol. 10978, pp. 59–71. Springer, Toledo, Spain (Jun 2018)
2. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. pp. 843–850. IFAAMAS, Budapest (2009)
3. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC). Lecture Notes in Computer Science, vol. 4294, pp. 339–351. Springer, Chicago (2006)
4. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**(3), 203–236 (2004)
5. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party sessions. Logical Methods in Computer Science **8**(1), 1–45 (Mar 2012)
6. 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 International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1054–1063. IFAAMAS, São Paulo (May 2017)
7. Desai, N., Singh, M.P.: On the enactability of business protocols. In: Proceedings of the 23rd Conference on Artificial Intelligence (AAAI). pp. 1126–1131. AAAI Press, Menlo Park (Jul 2008)
8. Ferrando, A., Ancona, D., Mascardi, V.: Decentralizing MAS monitoring with DecAMon. In: Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 239–248. IFAAMAS, São Paulo (May 2017)
9. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. Journal of the ACM **63**(1), 9:1–9:67 (2016)
10. Odell, J.J., Parunak, H.V.D., Bauer, B.: Representing agent interaction protocols in UML. In: Agent-Oriented Software Engineering. Lecture Notes in Computer Science, vol. 1957, pp. 201–218. Springer (2001)
11. Omicini, A.: SODA: societies and infrastructures in the analysis and design of agent-based systems. In: Proceedings of the International Workshop on Agent-Oriented Software Engineering. LNCS, vol. 1957, pp. 185–193. Springer (2000)
12. Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, chap. 5, pp. 107–135. Idea Group, Hershey, PA (2005)
13. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems **2**(4), 277–288 (Nov 1984)
14. Scribble: Scribble tools (Jan 2018), http://www.scribble.org
15. 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 (AAMAS). pp. 491–498. IFAAMAS, Taipei (May 2011)

16. Singh, M.P.: Semantics and verification of information-based protocols. In: Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). pp. 1149–1156. IFAAMAS, Valencia, Spain (Jun 2012)
17. Winikoff, M., Yadav, N., Padgham, L.: A new hierarchical agent protocol notation. Autonomous Agents and Multi-Agent Systems **32**(1), 59–133 (Jul 2017)
18. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 527–534. ACM Press, Bologna (Jul 2002)
19. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: Proceedings of the 8th International Symposium on Trustworthy Global Computing (TGC), Revised Selected Papers. Lecture Notes in Computer Science, vol. 8358, pp. 22–41. Springer, Buenos Aires (Aug 2013)