Castor and Pollux: A First Demonstration of a BSPL Protocol Discovery Tool

Matteo Baldoni[1 [0000−0002−9294−0408]], Cristina Baroglio[1 [0000−0002−2070−0616]], and Roberto Micalizio[1 [0000−0001−9336−0651]]

Università di Torino, 10149 Torino, Italy
{firstname.lastname}@unito.it

Abstract. Specifying interaction protocols for Multi-Agent Systems, especially with information-centric approaches like BSPL, is challenging, as designers must anticipate message structures, parameter roles, and acceptable interleavings. To simplify this task, we propose a framework for discovering BSPL protocols automatically from domain information and distribution control. We introduce Castor, a declarative language for modeling information elements, their sources, recipients, and aggregation into meaningful chunks. We pair Castor with Pollux, an automated synthesis tool that converts Castor models into BSPL protocols. Pollux treats protocol generation as a planning problem, where candidate messages are treated as actions, and BSPL semantic constraints are enforced during the search. Possible ambiguities are resolved via configurable heuristics, allowing the exploration of multiple safe protocol variants. Together, Castor and Pollux enable incremental, tool-assisted discovery of BSPL interaction protocols.

Keywords: BSPL Protocols · Protocol Discovery · CLIPS.

1 Introduction

Interaction is the key feature of Multi-Agent Systems (MAS) and distributed computing. It is the backbone through which agents achieve both their individual and social goals by coordination and information exchange. The good design of interaction protocols aims at achieving Software Engineering properties, among which the realization of loosely-coupled agents, and a clear separation of concerns between the business logic and the interaction logic.

Despite its importance, agent programming frameworks, like [4, 3], provide limited support for interaction protocol specification: specifically, they just provide *send* and *receive* primitives for message exchange, leaving the burden of implementing the interaction to the programmer. As Winikoff observes [12], these two primitives are overly low-level: they compel designers to frame interactions solely as reactive message exchanges (each sent in direct response to a prior receipt). This rigid approach over-constrains the interaction space, stifling agent autonomy and hindering their capacity for flexible, proactive engagement with others. Their use is analogous to the control transfer realized by

the `goto-label` instruction in imperative programming. Winikoff suggests that interaction should be modeled in more abstract terms.

An alternative is provided by the Blindingly Simple Protocol Language (BSPL) [10, 8, 7]. Rather than focusing on message flows, they claim the focus should be on the causal relationships among chunks of information to be exchanged. Rather than sending messages because some point in a fixed flow was reached, messages become *enabled* (to be sent) when certain information is available. Agents remain autonomous in deciding if and when an enabled message is to be sent.

Orpheus [2] illustrates how, through the BSPL lens, agent interactions can be streamlined by abstracting receive primitives within a middleware layer. This design empowers both the agent and the programmer to determine whether an enabled message should be sent, rather than enforcing a rigid logic. However, defining protocols in BSPL presents significant challenges. The framework's message parameters serve dual roles: they may function as payload, as coordinators (such as sequences or mutexes), or even as both simultaneously, complicating protocol specification. Langshaw [11] provides a first solution to overcome this issue. Langshaw is a declarative language for specifying protocols in terms of communicative actions, that adopts a centralized and synchronous perspective in order to simplify the modeling task, but a Langshaw model can be compiled into an asynchronous BSPL protocol.

In this work, we propose a different approach conceiving the protocol as *information+distribution*. Consequently, a BSPL protocol can be obtained as the result of a domain-independent procedure (the *distribution* part), which needs as input a conceptual model of the problem at hand (the *information* part). This second element is in general easier to define because it is only necessary to represent what is already known about the problem. To solve the equation, we introduce the twin components *Castor* and *Pollux*. Castor allows the identification of elementary pieces of information, where information is produced and who is interested in it. It also allows grouping elementary information into chunks, that are meaningful in the domain at issue. Pollux takes as input a Castor model and transforms it a search problem, aimed at building a BSPL protocol. Protocol construction is handled as a sort of planning task, where possible messages amount to basic actions. During the search, Pollux takes into account the constraints of the BSPL semantics and, in case of ambiguity (e.g., when the same information can be issued by different roles), it applies heuristics in order to produce safe and consistent protocols.

The paper is organized as follows. In Section 2 we shortly introduce BSPL and sketch its potential advantages from a Software Engineering perspective when compared to traditional message exchange. Sections 3 and 4 present our major contribution. In the former, we introduce the Castor language for conceptual information models, and give its semantics in Relational Algebraic terms. In the latter, we explain how Pollux discovers a BSPL protocol starting from a Castor model. Section 5 presents the tool we have developed, implementing the ideas of Castor and Pollux, and finally the paper concludes with some future directions in Section 6.

2 Background on Information Protocols and BSPL

This section provides background on BSPL and explains how the information-centric approach decouples interaction from agent-internal reasoning. As explained, BSPL defines interaction in terms of information flow rather than message exchange. Specifically, a BSPL protocol consists of a set of *roles* and a set of *message schemas*. Each message schema is defined by a name, a sender role, a receiver role, and a set of parameters. Some parameters are designated as *key parameters*: together they form a composite key that uniquely identifies protocol enactments. The others represent the information that may be exchanged during the interaction. Parameters are marked as ⌜in⌝ when their value must be known to the sender at the time of a message emission. Parameters are marked as ⌜out⌝ when must be unknown, and become known to the sender, as a result of sending the message. This explicit distinction between required and produced information allows BSPL to capture interaction constraints at a higher level of abstraction than message sequencing. A *message instance* in BSPL is a tuple of parameter bindings associated with a message schema. Bindings are annotated as incoming or outgoing, reflecting whether the sender already knows or produces the corresponding information.

Interaction constraints are based on the properties of information *causality* and *integrity*. Causality is enforced by requiring that all ⌜in⌝ parameters of a message instance be bound before the message can be sent. Integrity is enforced by constraining parameter bindings across message instances: no two message instances that share overlapping key parameter bindings may introduce different bindings for the same non-key parameter. These constraints ensure that interaction correctness emerges from information consistency rather than from adherence to predefined message orders. When every possible protocol enactment satisfies integrity, the protocol is said to be *safe*.

Crucially, these constraints are locally determinable. An agent can decide whether a message may be sent or accepted based solely on the information it currently knows, without requiring global synchronization or knowledge of the full interaction history. This property supports asynchronous communication and reduces coupling between agents.

A message instance is said to be *enabled* for an agent when: (i) all its ⌜in⌝ parameters are bound and known to the agent, and (ii) all its ⌜out⌝ parameters are unbound. An agent *knows* a binding for a parameter if it has previously sent or received a message containing that binding. Enablement, thus, depends exclusively on information availability, not on the prior exchange of specific messages.

Listing 1. Purchase Protocol.

```
FlexiblePurchase {
  role B, S
  parameter out ID key , out item , out status , out paid

  B -> S : Request [ out ID , out item ]
  S -> B : Shipment [ in ID , in item , out status]
  B -> S : Cash [ in ID , in item , out paid]
```

```
  B -> S : CreditCard [ in ID , in item , out paid]
}
```

Listing 1 illustrates a BSPL specification for a purchase protocol. The protocol defines two roles, B and S, and a set of message schemas. The key parameter ID uniquely identifies protocol enactments. Shared output parameters ensure that conflicting outcomes cannot be produced for the same enactment. For instance, it is not possible to generate both Cash and CreditCard messages for the same ID, as both messages would bind the same output parameter. The protocol is flexible since the ordering between Shipment and the payment (either by Cash or by CreditCard) is not established at design time, but determined during the actual enactment.

3 Castor: a Conceptual Information Modeling Language

Castor is a declarative language for specifying information models, characterized by multiple information sources. The goal of Castor is to capture how distributed information can be conceptually organized into complex chunks, that have a meaning on their own and that should be handled (communicated) as a whole. We explain it by relying on an E-business scenario, and we provide the language semantics in terms of relational algebra operators: a natural choice due to the information perspective of Castor.

The building block of a Castor model is the notion of *primitive information*: an indivisible piece of knowledge that is identified by a reference token, which consists of one or more identifier symbols. For instance, Listing 2 states that there exists an identifier `ID`, which is used to identify two primitive pieces of information: an item and a price.

Listing 2. Primitive information and identifiers.

```
 identifier ID.
 ID identifies item, price.
```

Semantics. Consider the Castor statements

identifier $id_1, \ldots, id_k$.
$id_1, \ldots, id_k$ identifies $p_1, \ldots, p_n$.

where $id_1, \ldots, id_k$ are identifiers, and $p_1, \ldots, p_n$ are primitive pieces of information. These statements amount to the creation of relations $T_{p_1}, \ldots, T_{p_n}$ each of which is defined over the schema $X_{p_i} = \{id_1, \ldots, id_k, p_i\}$. We introduce the notation:

1. $\kappa(p_i)$ to denote the set of identifiers $\{id_1, \ldots, id_k\}$;
2. $T_{p_i}(X_{p_i})$ to denote a relation T_{p_i} defined over the schema X_{p_i};
3. $\kappa(p)$ to denote $\cup_{i=1,\ldots,n}\kappa(p_i)$, where p is the list of primitive information $p_1, \ldots, p_n$.

As usual in relational algebra, the union of two schemas $X_{p_i} \cup X_{p_j}$ yields a new schema X defined over all the attributes in X_{p_i} and all the attributes in X_{p_j} without repetitions.

A primitive information unit is always *under the responsibility* of at least one role involved in the interaction, meaning that the role is a possible source of that primitive information. The only information that is relevant to an interaction protocol, is the information that is exchanged, therefore, for each primitive information there is always at least one role *that is interested in it*. Castor expresses these features as follows:

Listing 3. Responsibility and interest.

```
roles Buyer, Seller.
Seller responsible for ID, item, price.
Buyer interested in item, price.
```

The first line declares the list of roles in the protocol, while the next two respectively state that `Seller` can issue the primitive pieces of information `item` and `price`, and that `Buyer` is interested in them.

In an interaction context, a set of primitive pieces of information may acquire meaning on its own right, when considered altogether (building a complex *information chunk*). To this aim, Castor provides the construct `since`. For instance, the Castor listing

Listing 4. Complex information.

```
offer since item, price.
```

states that, when `item` and `price` are considered together, they acquire the meaning of `offer`. Offer can recursively be considered as a piece of information that contributes to building more complex chunks, as shown further below.

Semantics. Let t be the list $t_1, t_2, \ldots, t_n$, where t_i is a primitive or a complex information. We define the relation $T_t(X_t) = \bowtie_{i=1,\ldots,n} T_{t_i}(X_i)$, where $X_t = \bigcup_{i=1,\ldots,n} X_{t_i}$, and we extend the κ operator as follows $\kappa(t) = \cup_{i=1,\ldots,n} \kappa(t_i)$.

The statement c since t corresponds, in relational algebraic terms, to the creation of a new relation $T_c(X_c)$, such that $T_c(X_c) \subseteq T_t(X_t)$, and $X_c \equiv X_t$.

Notably, such semantics captures the fact that the schema of a complex relation is always expressed in terms of attributes occurring in simpler relations (with an endpoint in primitive information).

In some cases, there is the need to express the contextual acquisition of primitive pieces of information with the acquisition of some more complex information, they are part of. For instance, suppose that `offer` is the act by which a company advertises products sold at certain prices, when `offer` becomes known, also the involved `item` and `price` become known. In Castor this is captured as follows:

Listing 5. Complex information with bound statements.

```
offer since item, price.
item, price bound when offer.
```

The two statements should be read as "an offer is the pair item and price", and "item and price are known only with an offer".

Semantics. Consider the statement

```
p bound when c
```

where p is the tuple of primitive information $p_1, \ldots, p_n$ and c is defined by a statement of the form `c since` p, t., that is, p is part of the definition of c. Now, by definition $T_p(X_p) = \bowtie_{i=1,\ldots,n} T_{p_i}(X_{p_i})$, the statement *bound when* means that each of these primitive pieces of information are known only when c is known, which in relational algebraic terms means that whenever a new primitive information p_i is added into T_{p_i} then the same information is added in T_c, with the same values for the identifiers in X_c. Formally, $T_p(X_p) \bowtie T_c(X) \equiv T_p(X_p) \rtimes T_c(X_c)$, that is, for every tuple in $T_p(X_p)$ there exists a tuple in $T_c(X_c)$ that contains it.

Complex information can be used to define even more complex information. For instance, an offer can either be accepted or rejected, these two alternative choices represent each a new piece of information that needs to be captured. Castor allows the specification of mutual exclusion choices as follows:

Listing 6. Mutual exclusion.

```
accept since offer.
reject since offer.
accept xor reject.
```

The first two statements represent that `offer` is associated with an `accept` state and with a `reject` state but, due to the last line, these two states are mutually exclusive. This example clearly shows Castor's information-oriented perspective: the listing refines `offer` by adding a characterization that is meaningful in the E-business scenario without capturing a flow of messages. It is not a designer's goal to determine which role will take a decision, and which other role should be informed about it.

Semantics. Consider the statement c_1 xor c_2, where c_1, and c_2 are complex pieces of information. The two relations $T_{c_1}(X_{c_1})$ and $T_{c_2}(X_{c_2})$ must not share tuples, consequently:

1. if $\kappa(X_{c_1}) \cap \kappa(X_{c_2})$ is empty, then the two relations T_{c_1} and T_{c_2} do not share any tuples by construction;
2. otherwise, $T_{c_1}(X_{c_1}) \bowtie_{\kappa(X_{c_1}) \cap \kappa(X_{c_2})} T_{c_2}(X_{c_2}) \equiv \emptyset$, that is, for the common identifiers, there are not common tuples.

The last element of the Castor language is the `terminal` statement, which is used by Pollux to determine the criterion for ending the search. For instance, the following example

Listing 7. Terminal.

```
terminal shipment, reject.
```

Intuitively, such a statement declares that whenever one of the complex information in the list is produced, the end of a possible interaction path was reached. For simplicity, in the current solution we assume that these complex information are mutually exclusive, so the semantics of terminal is equivalent to that of `xor`: `terminal` $t_1, \ldots, t_n$ (where each t_i is complex) is equivalent to `xor` $t_1, \ldots, t_n$. In future extensions, we plan enrich this semantics allowing the designer to express terminal conditions as formulae.

A complete Castor model. We conclude this section with a complete Castor model for a simplified E-Business scenario.

Listing 8. The Castor model for a simplified E-Business scenario.

```
EBusiness {
    roles Buyer, Seller.

    identifier ID.
    ID identifies item, price, amount, receipt.

    Buyer responsible for amount.
    Buyer interested in item, price, receipt.

    Seller responsible for ID, item, price, receipt.
    Seller interested in amount.

    offer since item, price.
    item, price bound when offer.

    accept since offer.
    reject since offer.
    accept xor reject.

    payment since amount, accept.
    amount bound when payment.

    issue since receipt, payment.
    receipt bound when issue.

    shipment since payment.

    terminal shipment, reject.
}
```

In E-Business, a Seller and a Buyer engage in commercial transactions. The Seller may offer an item with its price, and the Buyer may either accept or decline

such an offer. Upon acceptance, the Buyer proceeds with payment, after which the Seller ships the item. Possibly, the Seller may issue a receipt upon payment. The Castor model captures these facts directly, without the need to anticipate what messages the two roles need to exchange. In the first eleven lines, in fact, the designer just declares the list of roles, and the list of primitive information together with the roles who are responsible for them and interested to them. These information emerge quite immediately from the case description. From line 13 to line 27, the designer specifies how primitive information relate to each other through the definition of complex information. Possible mutual exclusion constraints are again easily derived from the scenario description. Finally, on line 29, the designer specifies which complex information conclude a possible interaction. In all this process, the designer just focuses on the information dimension, without worrying about possible message schemas.

4 Pollux: A Tool for BSPL Protocol Discovery

A Castor model focuses on the primitive information that need to be shared, and how these information can be logically organized into meaningful piece of knowledge. However, Castor also models the principles of an information distribution problem. In fact, it also specifies which roles can produce primitive information and which others are interested in them. Pollux leverages these Castor constructs to transform a Castor model into a BSPL search problem. Pollux main steps are as follows:

1. Determine an initial set of causal dependencies among primitive information and message templates; during this step heuristics are used to discriminate between possible alternatives;
2. Set a BSPL search problem by identifying the initial state and the goal states;
3. Perform an exhaustive breadth-first search; the search assembles causal dependencies into possible BSPL messages, which are then used as operators for generating successor states;
4. Build up a BSPL protocol from the search tree gotten in the previous step.

Due to space limitations, we will not discuss each of these steps in details, but we will give the main intuitions underpinning each step, and then we will show some example of usage of the tool. In particular, Pollux is implemented by using the CLIPS rule-based system [9], so we will sketch some of the main facts that are produced and used during the search.

4.1 From Primitive Information to Causal Dependencies.

Castor models convey the causal dependencies existing among pieces of information, rather than specifying message structures. Pollux's first step is to extract these dependencies, and represent them as facts. Let's consider the following Castor statements:

```
c since p, t.
p bound when c.
```

where p is a list $p_1 \ldots, p_n$ of primitive information, while t is a list $t_1, \ldots t_m$ of primitive or complex information. From Castor's semantics we know that:

1. The information in p and t can be grouped together within a relation, and when this happens it acquires a new meaning, represented by c. Intuitively, c looks like a message, whose parameters are all the primitive pieces of information that are part of p and t.
2. Primitive information in p is `bound when` c, that is, it is known contextually with c. If c is interpreted as a message, then, p is a piece of information that is communicated for the first time within c. In BSPL terms, this corresponds to decorating as ⌜out⌝ the parameters in p.

These two points are represented in Pollux as the CLIPS facts:

```
(dep (id_dep IDD) (prec t) (cons p))
```

where IDD is a symbol used to identify the dependency relation, and prec and cons are multislots (i.e., lists of symbols) capturing the causal dependency: to say p (⌜out⌝ parameters) a role must know t (⌜in⌝ parameters). Of course, in case the bound-when clause is missing, the cons multislot is empty:

```
(dep (id_dep IDD) (prec t) (cons ))
```

Pollux extracts from a Castor model also information about possible senders and receivers of candidate messages corresponding to complex information. It does this job by looking at the roles that are responsible and interested to primitive information. In particular, when the list p of primitive information is not empty, Pollux pairs a dependency fact with the following message template:

```
(mlabel (label c) (id_dep IDD) (senders S) (receivers R))
```

where the slot label denotes the name of the message, id_dep is used to relate this fact to its corresponding dependency fact, and senders and receivers are multislots containing the lists of, respectively, roles that are responsible for all the primitive information in p (i.e., they can produce them), and roles that are interested in them.

In general, however, when p is empty, and we have only:

```
c since t.
```

the mlabel corresponding to c needs to be inferred since senders and receivers are not directly determined by a Castor model. To this end, Pollux simulates how complex information propagate among the roles. During this step, Pollux applies some heuristics in order to resolve possible conflicts. In particular, two scenarios can arise: 1) chain vs keep on; 2) reply vs forward.

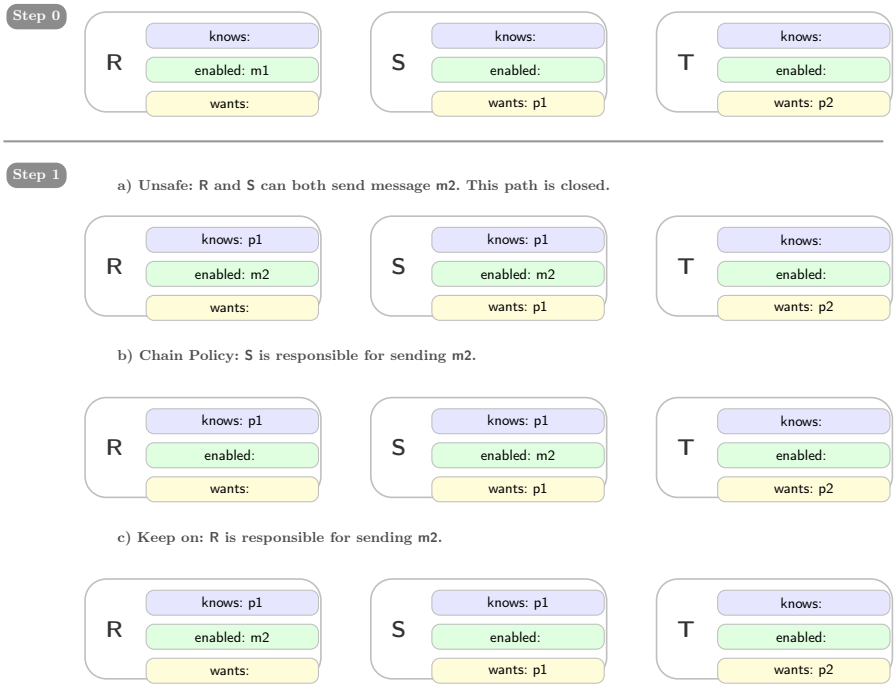

Fig. 1. Chain vs. Keep-on policies.

Chain vs. Keep-on Let us consider three roles R, S, and T. Assume that, at the beginning, the three roles do not know anything, and that role S is interested in primitive information p1, and T is interested in information p2. Let us now suppose that Pollux has determined, from a Castor model, that a possible message m1 has no ⌜in⌝ parameters, has ⌜out⌝ parameter p1, and can be sent from R to S. This situation is depicted in step 0 of Figure 1, which reports the local states of the three roles, that is, what each of them *knows*, the list of *enabled* messages for each of them, and the pieces of information they *want*.

Pollux simulates the possible situations at the subsequent step, that is, once message m1 is sent, and hence how the local states of R and S roles change. Let us assume that Pollux determines that message m2 has ⌜in⌝ parameter p1 and ⌜out⌝ parameter p2, which is of interest for role T. Now suppose that both roles R ans S are responsible for information p2, thus, both of them could send message m2 to T. Pollux explores three alternative situations, see step 1 in Figure 1:

(a) After sending message m1, both R and S know p1, and hence message m2 is enabled for both. This case, however, is *unsafe* as the same parameter p2 could be bound twice: Pollux discards this case.
(b) After sending m1, Pollux ascribes message m2 to role S and the following fact is inferred by Pollux:
(mlabel (label m2) (id_dep IDD-2) (senders S) (receivers T))

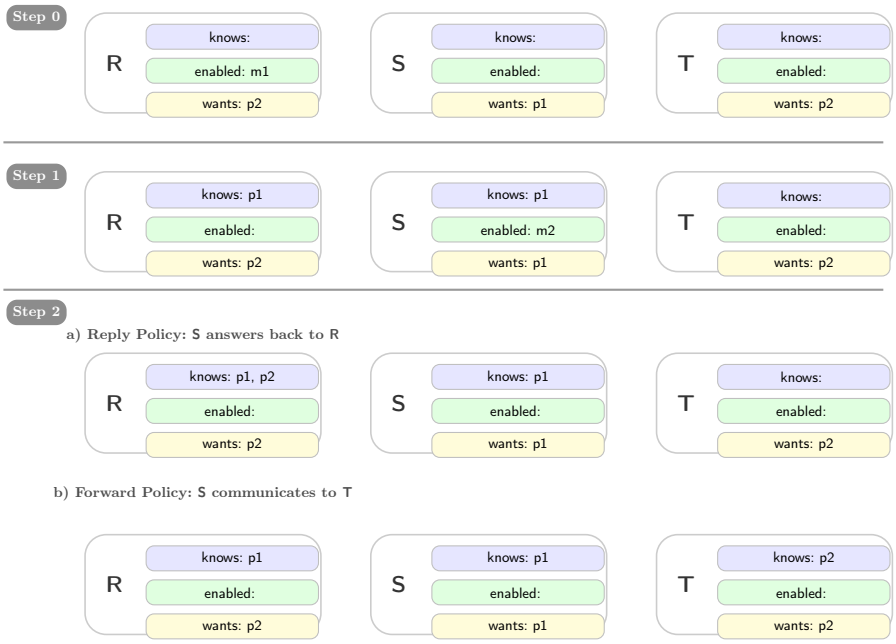

Fig. 2. Reply vs. Forward policies

Since the interaction is continued by the role player who received the previous message, we call this policy *chain*.

(c) After sending m1, Pollux ascribes message m2 to role R and infers the fact: (mlabel (label m2) (id_dep IDD-2) (senders R) (receivers T))
Since the interaction is continued by the role player who sent the previous message, we named this policy *keep-on*, in that the agent "keeps on talking".

By default, Pollux adopts the *chain* policy, but the user can change this setting from the Pollux GUI.

Reply vs. Forward. Let us consider again three roles R, S, and T that, at the beginning, do not know anything, and such that role S is interested in p1, whereas R and T are both interested in p2. In addition, we know that R is responsible for p1, while S is responsible for p2. As before, let us suppose that Pollux has determined, from a Castor model, that message m1 has no ⌜in⌝ parameters, has ⌜out⌝ parameter p1, and can be sent from R to S. This situation is depicted in step 0 of Figure 2, which reports the local states of the three roles.

Pollux simulates how the sending of m1 changes the local states of the roles to determine which other messages could be possibly enabled. In step 1 of Figure 2, it is shown the situation after the sending of m1. Specifically, both R and S know p1, and message m2 is now enabled in S (as before, message m2 has ⌜in⌝ p1 and ⌜out⌝ p2), but S can either send m2 to R or to T; these two scenarios are

depicted in step 2 of Figure 2, and also in this case Pollux needs to choose one of them:

(a) *reply* policy, S replies to R, the following fact is inferred:
(mlabel (label m2) (id_dep IDD-2) (senders S) (receivers R))
(b) *forward* policy, S sends a message to T, the following fact is inferred:
(mlabel (label m2) (id_dep IDD-2) (senders S) (receivers T))

By default, Pollux adopts the reply strategy, but also this setting can be changed via the Pollux GUI. In principle, S could send a message to both R and T, but the BSPL version currently handled by Pollux does not support multicast (see e.g., [5]). Of course, this is a possible future extension.

4.2 Discovering a BSPL Protocol

Once Pollux has determined the set of facts dep and mlabel capturing, respectively, the causal dependencies among information and the message templates, it starts a breadth-first search in order to find all possible interaction traces that lead from an initial state to a goal state. A state of the search captures an *interaction step*, consisting of roles' *local states*; that is, the information that each role knows at that particular step. At the beginning, all local states are empty. A goal state, instead, is any interaction step where at least one complex information c occurring in the terminal clause of the Castor specification has become known. In other words, some role has issued a message containing the primitive information composing c. This means that these information are known by some roles: they are kept in the local states of some roles.

The search simulates the progression of the roles' local states when messages, derived from mlabel and dep facts, are exchanged among the roles. More precisely, BSPL messages are generated according to the following rule.

```
1 (dep (id_dep IDD) (prec t) (cons p))
2 (mlabel (label l) (id_dep IDD) (senders S) (receivers R))
3 ⇒
4 S -> R : l [in t, out p, _l]
```

Where R and S are two roles, t is a list of either complex or primitive information, and p is a list of primitive information that are bound with the emission of message l. Note that a control parameter $_l$ is added by default to the list of ⌜out⌝ parameters of message l; it is also added as ⌜in⌝ parameter whenever role R needs the information tuple $\langle t, p\rangle$ for binding new parameters with fresh values while issuing a message. This control parameter, thus, allows Pollux to enforce message orderings when necessary. Control parameters introduced to capture specific Castor statements, below, are dealt similarly.

Pollux tries to merge messages whenever possible according to the rule

```
1 (dep (id_dep IDD−1) (prec t) (cons p1))
2 (dep (id_dep IDD−2) (prec t) (cons p2))
3 (mlabel (label l) (id_dep IDD−1) (senders S) (receivers R))
4 (mlabel (label m) (id_dep IDD−2) (senders S) (receivers R))
```

```
⇒
S -> R : l_m [in t, out p1,p2, _l, _m]
```

Namely, whenever two messages l and m share sender, receiver and ⌜in⌝ parameters, their ⌜out⌝ parameters can be merged into a new message. Also in this case appropriate control parameters $_l$, $_m$ are appended to the list of out parameters. Notably, the merged message l_m does not substitute the original messages l and m; rather, it represents a further possibility, clearly distinguished, that the human user can consider.

Pollux handles xor and terminal statements via control parameters. For instance, the mutual exclusion l xor m is assured by adding a common control parameter x-lm to the generated messages corresponding to the complex information l and m (and any other messages obtained by merging them):

```
Sl -> Rl : l [in tl, out pl, _l, x-lm]
Sm -> Rm : m [in tm, out pm, _m, x-lm]
```

The terminal statement, having the same semantics of xor, is treated similarly: l terminal m is captured by adding parameter x-terminal to the corresponding messages:

```
Sl -> Rl : l [in tl, out pl, _l, x-terminal]
Sm -> Rm : m [in tm, out pm, _m, x-terminal]
```

The search carried out by Pollux exploits the inferred message schemas and proceeds as follows:

- Initialize a queue with the initial interaction step where all the roles' local states are empty
- While the queue is not empty
  - Extract the first interaction step from the queue;
  - If this interaction step satisfies a goal condition, the path is closed, and the messages along this path from the initial state to the goal state are collected.
  - Otherwise:
    - Determine the set of message schemas enabled in the current interaction step;
    - For each enabled message schema, generate a successor interaction step by updating the current one: the sender's local state is modified by adding the ⌜out⌝ parameters, and the receiver's local state is modified by adding ⌜in⌝ and ⌜out⌝ parameters. Each successor state is appended to the queue

At the end of the algorithm, the BSPL protocol results from the messages collected along the visited paths leading to a goal state. The causality and integrity constraints of BSPL are preserved by relying on the added control parameters that either consent or prevent the enablement of message schemas during the search.

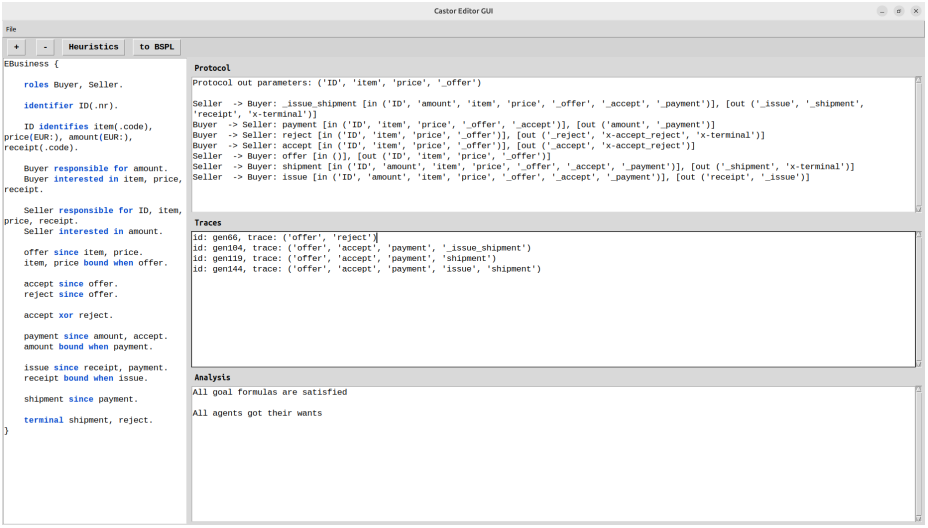

Fig. 3. The Castor&Pollux Tool.

5 The Castor&Pollux Tool

The Castor&Pollux Tool[1], implemented in Python, is sketched in Figure 3. The panel on the left allows editing a Castor model, then parsed by means of ANTLR 4.13.2[2], into an equivalent set of facts, that are given as input to Pollux. We implemented Pollux in CLIPS rule-based system. This choice is motivated by several reasons: 1) CLIPS is a declarative rule-based system that integrates also procedural coding, this allows the programmer to choose the most appropriate programming paradigm to cope with specific aspects of the implementation; 2) it offers a powerful pattern matching mechanism that helps writing compact querying rules about mlabel and dep facts; 3) it has a built-in breadth-first conflict resolution strategy. Pollux writes its outputs on the three boxes on the right panel. The first box shows the discovered BSPL protocol. The middle box shows all the found traces leading to a goal state. These traces help the user understanding whether the interaction progresses as expected. The last box shows, if any, goal states that were never achieved. This happens when Castor models are ill-defined, leading to unsafe BSPL protocols. Pollux detects unsafe messages during the search, and in those cases it closes those paths, but continues the search along other open paths. Thus, the protocols discovered by Pollux are safe by construction, although they might not find a path to each complex information in the terminal statement.

Figure 4 shows a snapshot for the Castor example in Listing 8. To ease readability, we report the discovered protocol in the following listing.

[1] Source code available at https://gitlab.di.unito.it/alice/pollux.
[2] https://www.antlr.org/

```
Protocol
Protocol out parameters: ('ID', 'item', 'price')

Seller  -> Buyer: _issue_shipment [in ('ID', 'amount', 'item', 'price', '_offer', '_accept', '_payment')], [out
('_issue', '_shipment', 'receipt', 'x-terminal')]
Buyer  -> Seller: payment [in ('ID', 'item', 'price', '_offer', '_accept')], [out ('amount', '_payment')]
Buyer  -> Seller: reject [in ('ID', 'item', 'price', '_offer')], [out ('_reject', 'x-accept_reject', 'x-terminal')]
Buyer  -> Seller: accept [in ('ID', 'item', 'price', '_offer')], [out ('_accept', 'x-accept_reject')]
Seller  -> Buyer: offer [in ()], [out ('ID', 'item', 'price', '_offer')]
Seller  -> Buyer: shipment [in ('ID', 'amount', 'item', 'price', '_offer', '_accept', '_payment')], [out
('_shipment', 'x-terminal')]
Seller  -> Buyer: issue [in ('ID', 'amount', 'item', 'price', '_offer', '_accept', '_payment')], [out ('receipt',
Traces
id: gen66, trace: ('offer', 'reject')
id: gen104, trace: ('offer', 'accept', 'payment', '_issue_shipment')
id: gen119, trace: ('offer', 'accept', 'payment', 'shipment')
id: gen144, trace: ('offer', 'accept', 'payment', 'issue', 'shipment')
```

Fig. 4. Protocol and Traces output boxes.

Listing 9. The BSPL protocol for the E-Business scenario in Listing 8.

```
EBusienss {
  role (B)uyer, (S)eller
  parameter out ID key, out item, out price, out amount, out receipt
  S -> B : offer [out ID, out item, out price, out _offer]
  B -> S : reject [in ID, in item, in price, in _offer, out _reject, out x-accept_reject, out x-terminal]
  B -> S : accept [in ID, in item, in price, in _offer, out _accept, out x-accept_reject]
  B -> S : payment [in ID, in item, in price, in _offer, in _accept, out amount, out _payment]
  S -> B : shipment [in ID, in item, in price, in _offer, in _accept, in _payment, out shipment, out _shipment, out x-terminal]
  S -> B : issue [in ID, in item, in price, in _offer, in _accept, in _payment, out issue, out _issue]
  S -> B : _shipment_issue [in ID, in item, in price, in _offer, in _accept, in _payment, out shipment, out issue, out _shipment, out _issue, out x-terminal]
}
```

The protocol contains messages that correspond to complex information of the Castor model, see e.g., offer, accept, payment, and others. However, Pollux has discovered that Shipment and Issue could be merged into a single message, identified by label as _issue_shipment. The BSPL schemas bring along several control parameters. Those starting with "_" are used for serializing messages when needed, for instance, since shipment must follow payment, parameter _payment is used as ⌜out⌝ in message Payment and ⌜in⌝ in message Shipment. Parameters starting with "x-" are used for expressing mutual exclusion constraints. For instance, parameter x-accept_reject is ⌜out⌝ both in accept and reject, therefore, only one of these two messages can actually be issued by Buyer. Of course, some of these control parameters might not be strictly required; for instance, parameter _issue could be removed with no effect on the protocol. These unnecessary parameters can be removed in a post-processing phase.

The second box of Figure 4 helps the designer understand what can actually occur during the interaction. The designer discovers, for instance, that message issue is optional: the interaction can be successfully completed even without it, see the third trace where it is missing. If this is not a desired behavior, the designer has to properly correct the Castor model by imposing, for instance, that the shipment can only be done after both payment and issue, as follows:

```
shipment since issue, payment.
```

The idea underlying the Castor&Pollux Tool is to assist a designer in the definition of a BSPL protocol. This means not only to fine tuning the Castor model in order to refine the resulting protocol. But, more importantly, progressively extend the Castor model by iteratively adding use cases, according to the current incremental strategies of code development. BSPL protocols incremental design, and its advantages with respect to the message-centric approach, are explained in [1].

6 Conclusion

This paper has introduced a tool for supporting the design of BSPL protocol. The tools consists of two main elements. The first is Castor, a high-level, declarative language, that focuses on the the information model (i.e., the set of relevant (complex) information and how they relate to each other), rather than on the messages. The rationale of Castor is that the information model is usually derived from a requirement specification of the problem at hand, and it does not require the designer to anticipate the possible set of messages, even in a simplified form like Langshaw [11]. The second, Pollux, implements a search algorithm for discoverying BSPL protocols starting from Castor models. The advantage of the Castor&Pollux framework is that the designer can define the protocol in an incremental way, by adding interaction cases one at a time in Castor, and assessing how these are translated into BSPL protocols by Pollux. The designer can set the heuristic policies so to explore alternative solutions, and selecting the one that best fits the scenario at hand.

In the present paper, Castor&Pollux handle the simplest version of BSPL, several extensions of which are discussed in the literature. For instance, Splee [5] introduces multicast, i.e., the possibility to send a message to a set of agents all playing a specific role. Pippi [6], instead, introduces the possibility of more articulated enactments by way of novel parameter adornments: ⌜nil⌝ permits a parameter not to be bound when a message is issued; ⌜any⌝ permits a parameter to either be ⌜in⌝ or ⌜out⌝ in a given enactment. Adding these features to Castor&Pollux is a natural extension of the present work.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Baldoni, M., Baroglio, C., Micalizo, R.: Evaluating the Benefits of Orpheus for Iterative and Incremental Development in MAS. In: Engineering Multi-Agent Systems (EMAS-26) (2026)
2. Baldoni, M., Christie V., S.H., Singh, M.P., Chopra, A.K.: Orpheus: Engineering Multiagent Systems via Communicating Agents. In: Walsh, T., Shah, J., Kolter, Z. (eds.) AAAI-25. pp. 23135–23143. AAAI Press (2025)
3. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. John Wiley & Sons, Ltd (2007)
4. Boissier, O., Bordini, R.H., Hübner, J., Ricci, A.: Multi-agent oriented programming: programming multi-agent systems using JaCaMo. MIT Press (2020)
5. Chopra, A.K., Christie, S., Singh, M.P.: Splee: A Declarative Information-Based Language for Multiagent Interaction Protocols. In: Larson, K., Winikoff, M., Das, S., Durfee, E.H. (eds.) Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017. pp. 1054–1063. ACM (2017)
6. Chopra, A.K., Christie, S., Singh, M.P.: Pippi: Practical protocol instantiation. In: Faliszewski, P., Mascardi, V., Pelachaud, C., Taylor, M.E. (eds.) 21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022. pp. 281–289. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS) (2022)
7. Christie, S.H., Chopra, A.K., Singh, M.P.: Bungie: Improving fault tolerance via extensible application-level protocols. Computer **54**(5), 44–53 (2021)
8. Christie, S.H., Chopra, A.K., Singh, M.P.: Mandrake: multiagent systems as a basis for programming fault-tolerant decentralized applications. Autonomous Agents and Multi-Agent Systems **36**(1) (2022)
9. Riley, G., Giarratano, J.: CLIPS A Tool for Building Expert Systems. https://www.clipsrules.net/, accessed: 2026-02-22
10. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In: Sonenberg, L., Stone, P., Tumer, K., Yolum, P. (eds.) 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011). pp. 491–498 (2011)
11. Singh, M.P., Christie V., S.H., Chopra, A.K.: Langshaw: Declarative Interaction Protocols Based on Sayso and Conflict. In: Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024. pp. 202–210 (2024)
12. Winikoff, Michael: Challenges and Directions for Engineering Multi-agent Systems. CoRR (2012), http://arxiv.org/abs/1209.1428