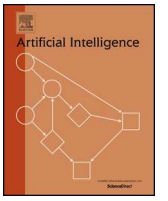
Artificial Intelligence 348 (2025) 104398

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Argus: Programming with communication protocols in a belief-desire-intention architecture

Samuel H. Christie V [a], Munindar P. Singh [a], Amit K. Chopra [b,*]

[a] North Carolina State University, Raleigh, NC 27695, USA
[b] Lancaster University, Lancaster, LA1 4WA, UK

A B S T R A C T

Protocols model multiagent systems (MAS) by capturing the communications between its agents. Belief-Desire-Intention (BDI) architectures provide an attractive way for organizing an agent in terms of cognitive concepts. Current BDI approaches, however, lack adequate support for engineering protocol-based agents.

We describe Argus, an approach that melds recent advances in flexible, declarative communication protocols with BDI architectures. For concreteness, we adopt Jason as an exemplar of the BDI paradigm and show how to support protocol-based reasoning in it. Specifically, Argus contributes (1) a novel architecture and formal operational semantics combining protocols and BDI; (2) a code generation-based programming model that guides the implementation of agents; and (3) integrity checking for incoming and outgoing messages that help ensure that the agents are well-behaved. The Argus conceptual architecture builds quite naturally on top of Jason. Thus, Argus enables building more flexible multiagent systems while using a BDI architecture than is currently possible.

1. Introduction

We address the problem of building decentralized multiagent systems, whose member agents are autonomous and heterogeneous. Decentralization is a crucial requirement in any application setting in which more than one stakeholder is involved because the agents representing and acting on behalf of different stakeholders must be able to (1) act independently and flexibly, thereby indicating their autonomy [1], and (2) be designed and configured independently, thereby indicating their heterogeneity [2].

With these concerns at heart, interaction-oriented approaches have historically emphasized agent communication [3], especially *protocols* [4,5], including higher-level abstractions such as commitments, as models of multiagent systems [6,7]. Broadly, a protocol specifies the constraints on messaging between agents. The main benefit of a protocol as a basis for a multiagent system is that it enables the implementation of each agent in the system independently of other agents, based solely on the roles it plays in the protocol. In other words, a protocol supports implementing autonomous and heterogeneous agents by capturing the extent of the coupling between them. A second benefit of protocols is that they can be verified to help design suitable interactions for a multiagent system [8,9] without being overwhelmed with the internal details of the agents in the multiagent system being designed.

In this paper, we turn to the problem of implementing *cognitive* agents on the basis of protocols. Agent-oriented approaches in the cognitive tradition, such as Jason [24,10], help specify autonomous agents in terms of their beliefs and intentions. These abstractions can capture stakeholder requirements succinctly and naturally by matching folk psychological constructs that stakeholders would understand and avoiding implementation details [11]. The agent-oriented approaches are undergirded by formal semantics and supported by programming frameworks that enable a programmer to specify the operations of an agent in terms of beliefs and

* Corresponding author.
E-mail addresses: schrist@ncsu.edu (S.H. Christie), singh@ncsu.edu (M.P. Singh), amit.chopra@lancaster.ac.uk (A.K. Chopra).

https://doi.org/10.1016/j.artint.2025.104398
Received 26 June 2024; Received in revised form 23 July 2025; Accepted 24 July 2025

Available online 6 August 2025

intentions such that the agent's behavior is directly linked to its cognitive makeup. Existing cognitive agent programming approaches support communication between agents; however, in general, they haven't kept up with advances in modeling interactions.

In a nutshell, both agent-oriented and interaction-oriented approaches aim to simplify the engineering of multiagent systems via programming abstractions. Further, they are mutually complementary. Whereas the cognitive agent-oriented approaches can be seen as capturing the internal reasoning that drives an agent's interactions, the interaction-oriented approaches may be seen as placing constraints on such reasoning for purposes of interoperability. However, as of yet, there is no approach that combines the two themes in a cohesive approach for building multiagent systems.

1.1. The Argus approach

This paper tackles the above challenge. It synthesizes the agent-oriented and interaction-oriented approaches into a new approach for developing multiagent systems. Our proposed approach is to replace the agent communication part of a BDI architecture with protocols. Specifically, we leave the BDI reasoning engine and the core abstractions of beliefs, goals, and plans unchanged. That is, we do not demand any change to whatever operational semantics holds for the BDI reasoning except for what concerns communication. Thus, agents can be authored and executed much as before, albeit with streamlining due to the proposed communication semantics and protocol handling.

We adopt Jason as the exemplar of the cognitive tradition because of its well-deserved prominence in agent programming and its extensive support for communications. Jason is an exemplary BDI framework that provides rule-based programming based on beliefs and intentions to build BDI agents. Jason agents can communicate with each other based on their beliefs and intentions, and thereby realize multiagent systems. Jason's communication model and semantics, however, suffer from fundamental limitations: They provide predetermined performatives (message or "speech act" types) and couple agents unnecessarily. Jason's strong treatment of communications makes it an ideal target for our investigation for two reasons. One, Jason provides a concrete model of communications in a traditional BDI approach that enables us to demonstrate the benefits of introducing interaction orientation. Two, because Jason has a communication model, it provides us with an engineering challenge to replace the communications part of it.

In contrast, other BDI approaches, e.g., CAN [12], do not pay special attention to communication, potentially treating it like any action. Incorporating communication protocols in those approaches is conceptually unproblematic since there is nothing that has to be replaced. Likewise, they provide less of an opportunity to show the benefits of an interaction-oriented approach to communication vis à vis a BDI approach.

We adopt information protocols [13] as an exemplar of the interaction-oriented tradition. Information protocols capture interactions between agents abstractly and seek to maximally decouple the agents—that is, couple the agents only to the extent necessary for interoperation in the desired multiagent system. The associated abstractions deal with information transfer as a basis for causality and integrity, which as well are undergirded by a formal semantics and associated with a programming framework. Information protocols may be composed and verified for properties such as liveness and safety, thus enabling the verification of a multiagent system before implementing agents to play roles in it [14]. They can be enacted flexibly and asynchronously without requiring message ordering guarantees from the underlying communication service [15]. Information protocols enable specifying and implementing fault tolerance at the application level [16]. In addition, they enable precise specification of commitments and other norms [17,18].

Because the concrete realization of our approach relies on Jason and contributes to the Jason ecosystem, we name our approach *Argus*—in Greek mythology, the builder of the Argo, the ship that transports Jason on his quest.

1.2. Contributions

Our overarching contribution is to unite BDI-oriented agent programming and information protocols to simplify implementing loosely coupled agents, and demonstrated concretely via Argus, as stated above.

Argus supports a reactive model to map information-based communication to BDI reasoning. The Argus operational semantics makes weak assumptions about the infrastructure, preserves autonomy (agents may ignore or respond to messages as they see fit), and preserves heterogeneity (separating local and internal states and avoiding hidden dependencies). Argus makes few assumptions about the underlying agent reasoning and may be readily adapted to other cognitive agent-programming languages, such as 2APL [19], 3APL [20], CAN [12], and GOAL [21].

We identify and resolve limitations of Jason's support for communications: no support for protocols; adoption of Knowledge Query and Manipulation Language (KQML) [22] primitives and semantics; selection and social acceptability functions to control communications that are neither public nor based in beliefs and plans; and inadequate support for autonomy and heterogeneity. KQML was a landmark contribution to agent communication, but there is no good reason to confine agent programming to KQML over thirty years after its inception. Our objective here is not to criticize Jason or KQML but to bring forth shortcomings of Jason's treatment of agent communication to show the need for improvement.

It is worth noting that implementing an agent that is compliant with the protocol and exploits its flexibility is a nontrivial activity. A flexible protocol can, in the worst case, have enactments that are exponential in the size of its specification. To support the task of implementing an agent, Argus bundles a code generator that produces Jason code that captures the changes to an agent's view of the protocol state and guides the implementation.

1.3. Running example: academic testing

We consider a typical US university setting, which involves professors, teaching assistants (TAs), and students. In our example, a course is taught by one professor, who is assisted by one TA for that course. Students enroll in one or more courses. Each course has a test that the professor prepares along with a grading rubric (instructions) for the TA. Each student enrolled in that course prepares a solution to the test.

We focus on the interactions here. A professor begins a test and asks students to provide solutions to a series of questions. The professor sends the grading rubric to the TA. A student submits their solutions to the TA, who grades them according to the rubric and sends the grades to the professor.

2. Background: agent programming in Jason

Jason is an extended implementation of the AgentSpeak logic-programming language for specifying agent behavior [24]. In Jason, an agent is modeled as having *beliefs*, which capture the state of the world; *goals*, which capture its objectives; and *plans*, which are methods for realizing its goals. To facilitate building multiagent systems, Jason adopts communication primitives based on KQML.

To illustrate Jason's programming model, especially how it weaves together communication and reasoning in an agent, Listing 1 and Listing 2 give snippets of professor Pnin and student Lancelot's implementations in Jason.

Listing 1: Jason snippet of Pnin, the agent who plays PROFESSOR.

```
1  student(s1, lancelot).
2  student(s2, galahad).
3  question(q1, "What is your name?").
4  question(q2, "What is your quest?").
5  !start.
6
7  +!start <-
8    for (student(TID, Student)) {
9      .send(Student, tell, begin_test(TID));
10     for (question(QID, Q)) {
11       .send(Student, tell, challenge(TID, QID, Q));
12       .print("challenge", Student, TID, QID, Q);
13     };
14   }.
```

The first two lines of Listing 1 add beliefs that there are two students, Lancelot and Galahad, with student IDs `s1` and `s2`, respectively. The next two lines add beliefs for questions, each composed of an identifier and a string stating the questions. Line 5 adds an initial goal to achieve *start*.

Lines 7–14 describe a plan for achieving the *start* goal: for each student, send a begin test message containing just the TID, and then send each question. Note that functions (plans), loops, and procedures (e.g., *.send*, which is built-in) work via parameter unification. For example, the for loop works by finding all terms that match the provided structure, using capitalized names (e.g., TID and Student) to represent variables. The loop body is executed once for each match; in this case, once for each student, with *(TID, Student)* bound to *(s1, lancelot)* and *(s2, galahad)* successively. The *.send* procedure takes three parameters: the name of the agent receiving the message (the *Student*), the speech act to perform (*Tell*), and the term to send (the challenge). The *Tell* speech act adds the term to the recipient's belief base (and triggers a matching plan if there is one).

Listing 2: Jason snippet of Lancelot, an agent who plays STUDENT.

```
1  +challenge(TID, QID, Q) <-
2    .print("I have been given challenge:", Q).
```

Lines 1–2 of Listing 2 give a plan for student Lancelot that reacts to the addition of the *challenge(TID, QID, Q)* belief. This belief is added automatically when the *Tell* is received, triggering the plan. The specific implementation simply prints information about the question the student was given.

An agent can have multiple plans for achieving a goal. If so, it tries to select the first one that is applicable, based on the *guards* of the plan.

Listing 3: Guard for challenge plan.

```
1  +challenge(TID, QID, Q)
2    : student(TID, lancelot)
3    <- ...
```

Line 2 in Listing 3 shows the plan for `challenge` extended with a single guard. This guard checks for the existence of the belief `student(TID, lancelot)`—in other words, it checks that Lancelot's agent believes TID is his student ID. If Lancelot's agent has a matching belief, it will select and execute this plan. Otherwise, it will skip over this plan and pick another one if available.

Table 1
Performatives supported by Jason.

Performative	Meaning
tell ⟨belief⟩	recipient should add ⟨belief⟩
untell ⟨belief⟩	recipient should remove ⟨belief⟩
achieve ⟨goal⟩	recipient should achieve ⟨goal⟩
unachieve ⟨goal⟩	recipient should drop the intention of achieving ⟨goal⟩
tellHow ⟨plan⟩	recipient should add ⟨plan⟩
untellHow ⟨plan⟩	recipient should disregard ⟨plan⟩
askIf ⟨query⟩	recipient should reply if ⟨query⟩ is true
askAll ⟨query⟩	recipient should reply with all answers matching ⟨query⟩
askHow ⟨query⟩	recipient should reply with plans matching ⟨event⟩

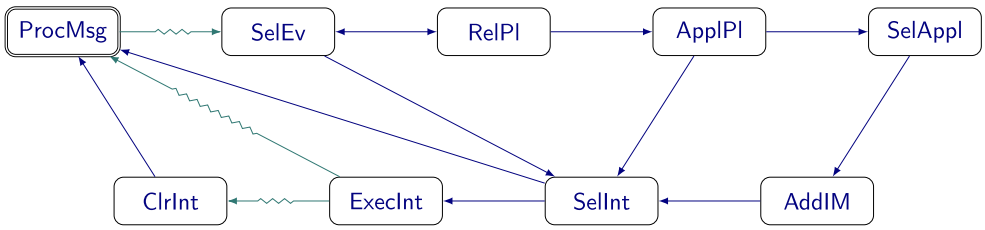


Fig. 1. The Jason reasoning cycle, highlighting the transitions pertaining to communication with zig-zag arrows. The other transitions arise from internal reasoning (processing of beliefs, plans, and intentions) in Jason.

2.1. Communication

Communication in Jason primarily uses the `.send` function, though others such as `.broadcast` also exist, and the language can be extended with custom plugins.

The syntax of sending a message is `.send(Recipient, Performative, Content)`. Here, Recipient is a reference to the agent the Jason runtime should deliver the message to. Performative is the name of the speech act (i.e., illocutionary act [23]) this message is performing. Content is a literal being sent as the body of a message and refers to the content of the message, i.e., a belief that the recipient should adopt. Listing 1 uses *Tell* as the performative, which means that the receiving agent should adopt the content of the message as a belief. The list of supported performatives is given in Table 1.

2.2. Jason reasoning cycle

Fig. 1 shows the Jason reasoning cycle, involving the various processing steps and how a Jason reasoner may transition from one step to the next. The entire set of processing steps in Jason is {ProcMsg, SelEv, RelPl, ApplPl, SelAppl, AddIM, SelInt, ExecInt, ClrInt} [24, pp. 234–235]. These labels stand for, respectively: processing a message from the agent's mail inbox, selecting an event from the set of events, retrieving all relevant plans, checking which of those are applicable, selecting one particular applicable plan (the intended means), adding the new intended means to the set of intentions, selecting an intention, executing the selected intention, and clearing an intention or intended means that may have finished in the previous step.

In Jason, a receiver processes a message from its inbox when (1) its programmer-specified selection function S_M selects that message, and (2) the constraints specified in the SocAcc function are met. Processing the message means inserting it into the receiver's belief base. (See Jason's TELL rule [24, p. 244].) The motivation behind S_M is to capture the priority of a message. The motivation behind SocAcc is to capture reasoning based on social constraints such as power, trust, and so on.

3. Information protocols

An information protocol, as specified in the Blindingly Simple Protocol Language (BSPL) [13], specifies communication in a multiagent system and provides a basis for implementing its flexible agents in a loosely-coupled manner. Listing 4 illustrates the main features of BSPL via our running example.

Listing 4: The grading protocol.

```
1 Grading {
2   roles Professor, Student, TA
3   parameters out TID key, out QID key, out Grade
4   private Question, Solution, Answer
5
6   Professor ↦ Student: beginTest[out TID key]
7   Professor ↦ Student: challenge[in TID key, out QID key, out Question]
```

```
8
9    Professor ↦ TA: rubric[in TID key, in QID key, out Solution]
10   Student ↦ TA: response[in TID key, in QID key, in Question, out Answer]
11   TA ↦ Professor: result[in TID key, in QID key, in Answer, in Solution, out Grade]
12 }
```

A protocol specifies the *roles* that participate in it. It also specifies the *message schemas* of the messages to be sent and received by the (agents playing the) roles. A message schema has a name, a sender role, a receiver role, and one or more parameters, some of which are designated ⌜key⌝ and each of which is adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝. A *message instance* is a tuple of bindings for the ⌜in⌝ and ⌜out⌝ parameters of that schema (⌜nil⌝ parameters have no bindings). The ⌜key⌝ parameters of a schema form a composite key and uniquely identify its instances.

Parameter adornments capture causality. An agent's *local state* is the set of messages it has observed, that is, sent or received. To emit an instance of a schema, parameters adorned ⌜in⌝ must have bindings in the sender's local state, and parameters adorned ⌜out⌝ and ⌜nil⌝ must not have bindings in the sender's local state. We say a parameter binding is *known* to the agent if its binding exists in the local state. Thus, in simple terms, to emit an instance, the ⌜in⌝ parameters' bindings must be known already to the agent, and the ⌜out⌝ and ⌜nil⌝ parameters' bindings must not be known already. Upon emission of the instance, it becomes part of the local state, and the ⌜out⌝ parameters bindings become known; the ⌜nil⌝ parameters remain unknown.

By uniqueness, no two message instances with the same bindings for overlapping ⌜key⌝ parameters may have distinct bindings for common non-key parameters. Since bindings are introduced through ⌜out⌝ parameters, no two message instances may have overlapping key parameter bindings as well as a binding of the same ⌜out⌝ parameter. BSPL thus captures *causality* and *integrity* through information.

How may an agent create message instances for emission? The bindings for the ⌜in⌝ parameters must obviously come from the local state. The bindings for the ⌜out⌝ parameters must, however, be *generated* by the agent via internal reasoning. For example, say Pnin (an agent playing the role PROFESSOR) wants to send a *challenge*. Pnin's internal reasoning for generating Question may involve looking up a database of questions from which it selects a question with a suitable difficulty level and, moreover, has not been used in *challenge*s in the last three years.

TID identifies the test being taken. Since TID and QID are both marked ⌜key⌝, where they appear together, they constitute a composite key and jointly identify a challenge within a test. That is, each test may have multiple challenges, one for each binding of QID. To send *result*, for any ⟨TID, QID⟩ tuple, an agent playing the TA role needs to know Solution and Answer. This use of key parameters illustrates correlation and joining of information from different roles.

Notably, a message may be received at any time, that is, in any relative order with respect to other messages, obviating the need for ordered-delivery communication services. For example, the information needed for TA to send *result* comes from receiving *response* from STUDENT and *rubric* from PROFESSOR. These messages may be received in any order by TA.

Grading can be enacted flexibly. After beginning a test, *Professor* may send challenges and rubrics in any order. Moreover, they may be received by STUDENT and TA in any order. STUDENT may respond to challenges received in any order and TA may grade responses (for which it has also received rubrics) in any order. Fig. 2 demonstrates an enactment for TID `t1` and QIDs `q1` and `q2` (other parameters are elided).

The formal syntax of our language, based on BSPL [13], is given in Table 2, and described in detail below. A superscript of + indicates one or more repetitions, superscript ∗ indicates zero or more, and ⌊ and ⌋ delimit expressions, which are optional when without a superscript.

L_1. A specification document consists of one or more protocols (which may be individual messages).

L_2. A protocol declaration consists of a name, roles, a public parameter expression, optional private parameters, and references to constituent protocols or messages. The public parameters with the key qualifier form this declaration's key.

L_3. A parameter expression is a comma-separated list of parameters.

L_4. A parameter has an adornment and name and may be optionally declared key.

L_5. A reference to a protocol consists of the name of the referenced protocol and a parenthesized parameter expression matching the protocol's declaration.

L_6. A message schema consists of a name, a sending role, a receiving role, and a parameter expression.

L_7. An adornment is either ⌜in⌝, ⌜out⌝, or ⌜nil⌝.

Together, ⌜in⌝, ⌜out⌝, and ⌜nil⌝ capture all the possibilities from the point of view of what knowledge an agent has and what knowledge it may generate through its actions (message emissions). These are either *know* (⌜in⌝) or *don't know* (⌜out⌝ or ⌜nil⌝). *Don't know* can be further broken down into *can generate* (⌜out⌝) and

cannot generate (⌜nil⌝). The parameter adornments may be used to constrain the ordering and occurrence of an agent's actions with respect to other actions. For example, if we wanted to ensure that an agent could only emit message m after observing (sending or receiving) another message m', then we would have some parameter that was ⌜out⌝ only in m' and ⌜in⌝ in m. If we wanted ensure that a message m could not occur after m' had been observed, then we would have some parameter that was either ⌜in⌝ or ⌜out⌝ in m' and ⌜nil⌝ in m.

4. The Argus architecture

Argus is a programming model for agents based on information protocols. The programming model takes a protocol as an input and provides abstractions that make it convenient to independently implement agents.

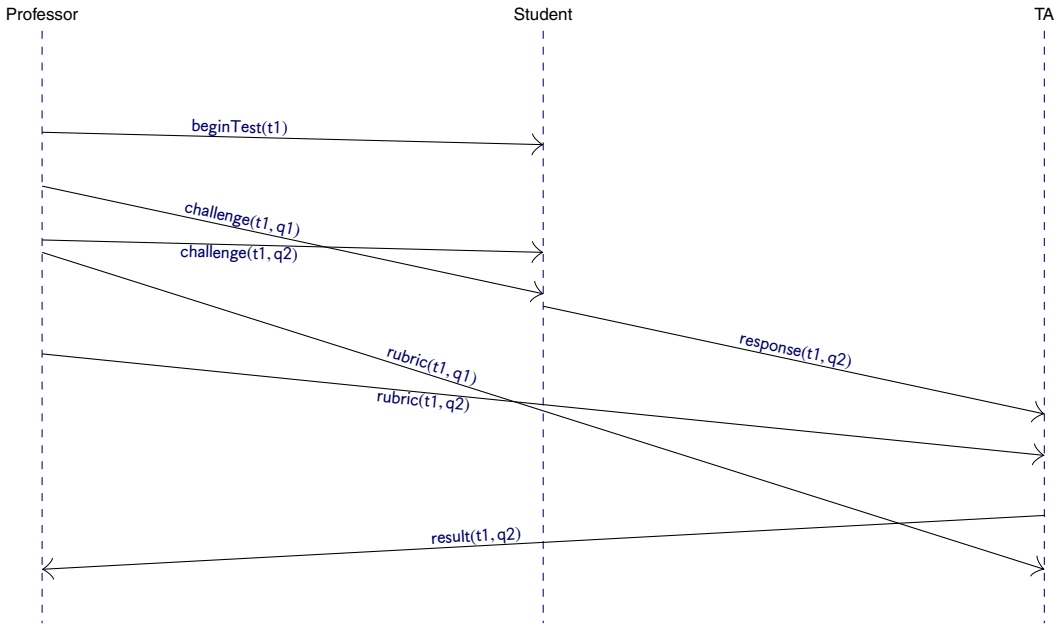


Fig. 2. An enactment of *Grading* demonstrating the flexibility afforded by information protocols.

Table 2
BSPL Syntax.

L_1	Spec	$\longrightarrow \lfloor Protocol \vert Message \rfloor^+$
L_2	Protocol	$\longrightarrow$ *Name* { roles $Name^+$ public *ParamExpr* ⌊private *ParamExpr*⌋ $Reference^+$ }
L_3	ParamExpr	$\longrightarrow$ *Parameter* $\lfloor$, *Parameter*$\rfloor^+$
L_4	Parameter	$\longrightarrow$ *Adornment Name* ⌊key⌋
L_5	Reference	$\longrightarrow$ *Name*(*ParamExpr*) | *Message*
L_6	Message	$\longrightarrow$ *Name* $\mapsto$ *Name*: *Name*[*ParamExpr*]
L_7	Adornment	$\longrightarrow$ in | nil | out

We describe the main elements of the Argus architecture. Let Pnin and Timofey be agents playing the PROFESSOR and TA roles, respectively. Let Galahad and Lancelot be agents playing the STUDENT role.

Fig. 3 shows a multiagent system under Argus. BDI Agents interact on the basis of an information protocol. An agent's local state comprises beliefs corresponding to the messages it has observed (sent or received) and is used for validating messages before emission and after reception. The internal state comprises beliefs about whatever is relevant to the agent's reasoning besides what is included in the protocol. That is, the internal state is separate from the local state. There are no other beliefs. The adapter applies the protocol specification to validate both incoming and outgoing messages and update the local state. Each agent has plans and a BDI reasoner that executes its plans. An agent's control state is given by its current intentions and associated objects.

Some of the plans are generated by our tooling based on the protocol specification. These plans are essentially to emit messages. Each plan is triggered by the addition of a belief corresponding to a message observation and specifies the state the enactment should be in for a further message to be emitted. These plans offer an opportunity for the developer to plug in internal reasoning satisfying which the message will be emitted. Such an emission plan is necessarily incomplete because the internal logic is necessary to produce the bindings of the message's ⌜out⌝ parameters. The internal reasoning could itself involve other plans written by the agent developer.

4.1. Representing the local state

Argus preserves agent autonomy by separating each agent's local state (containing information shared between agents who communicate with each other) from its internal state (containing private information).

In Argus, a valid message observation (emission or reception) is represented as a belief and added to the local state. For example, referring to Listing 4, the message

challenge[s1, q1, "What is your name?"]

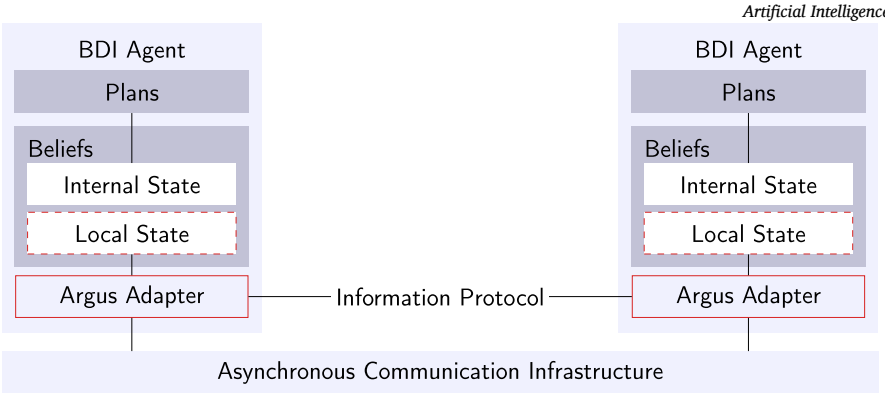


Fig. 3. The Argus architecture is realized on top of the BDI architecture in that Argus addresses how BDI agents interact. Each agent has an Argus adapter generated for the roles it plays in the information protocol. The adapter validates incoming and outgoing messages against the protocol and updates the local state. Argus blends into the BDI architecture because the Argus local state is nothing more than a set of beliefs, as are already available to the BDI reasoner in the BDI architecture.

from Pnin to Lancelot, if it passes validation, is constructed as the Jason term

challenge("Pnin", "Lancelot", s1, q1, "What is your name?")

and is added to Pnin's local state upon emission and to Lancelot's local state upon reception.

4.2. Handling message observations

To avoid conflict with Jason's builtin procedure .send, we postulate (and implement) new builtin procedures .emit and .emitAll.

Executing .emit for some message instance causes the adapter to validate the instance against the local state. Specifically, the bindings for the ⌜in⌝ parameters must be already known (from the local state); and the bindings for the ⌜out⌝ and ⌜nil⌝ parameters must not already be known. If the validation is successful, the adapter adds the instance to the local state (as described above) and transmits the message instance to the recipient via the appropriate channel. The recipient can be specified directly as a string address in "ip:port" format, or will be loaded from the adapter configuration.

Similarly, for .emitAll, a multicast primitive, the same message is emitted in every MAS that the sender is already participating in. This means that for Pnin, who is configured with two MASs, one for Lancelot and one for Galahad, he can send the questions to both students with a single .emitAll.

Listing 5: Partial Professor Pnin in Argus.

```
1  student("lancelot", "Lancelot")
2  student("galahad", "Galahad").
3  question(q1, "What is your name?").
4  solution("lancelot", q1, "Sir Lancelot of Camelot").
5  solution("galahad", q1, "Sir Galahad of Camelot").
6  !start.
7
8  +!start <-
9    TID = "midterm";
10   .print("Starting test ", TID);
11   .emitAll(begin_test(MasID, Professor, Student, TID));
12
13   for (question(QID, Question)) {
14     .print("Challenge ", QID, ": ", Question);
15     .emitAll(challenge(MasID, Professor, Student, TID, QID, Question));
16     for (student(MasID, Student)) {
17       solution(MasID, QID, Solution);
18       .emit(rubric(MasID, Professor, TA, TID, QID, Solution));
19       .print("Solution for", MasID, QID, "is", Solution);
20     };
21   }.
```

For example, in Line 15, Pnin sends *challenge* to all students by invoking .emitAll. Suppose TID is "midterm" and QID is q1, and Question is *"What is your name?"*. Pnin's adapter constructs the corresponding message instance and validates it. If the instance passes validation, it adds the assertion *challenge("lancelot", "Pnin", "Lancelot", "midterm", q1, "What is your name?")* to the local state and sends the message instance to Lancelot, with a similar message for Galahad.

Each interaction happens within the context of MAS. In our implementation, borrowing from Kiko [25], the variable MasID identifies the MAS. In Listing 5, the MasID corresponds to the name of the student in lowercase, e.g., `lancelot` (line 1). In effect, each student is in a different MAS but with the same Professor and the same TA.

When the adapter receives a message instance and finds it consistent with the local state, it inserts the message into the local state. Given the information model, consistency simply means that there would be no key violation in the local state if that message were inserted. For example, when Lancelot receives the above message, Lancelot's adapter checks the local state to verify that no message with the same TID and QID but a different Question has been observed. If the check passes, then

challenge("lancelot", "Pnin", "Lancelot", s1, q1, "What is your name?")

is added to Lancelot's local state.

It is worth emphasizing the distinction between an agent's local state and internal state. Lines 1–5 model Pnin's question bank. The questions and their solutions are part of Pnin's internal state.

Message parameters exclusively apply to the local state; their bindings don't become part of the local state until a message containing them is recorded in the local state. Whether a message may be emitted depends only on the local state, not the internal state. The *rubric* message has ⌜out⌝ Solution. Pnin's emission of this message (Line 18) is not blocked merely because he has the solutions in his internal state. A solution in the internal state doesn't become a Solution in Pnin's local state until he sends a *rubric* for some TID, QID combination with the solution as the binding for Solution. Once Pnin has sent it, Solution becomes part of the his local state relative to that TID, QID combination and he is blocked from sending any further *rubric* messages for the combination.

Multicast messages are handled by generating separate messages for each MAS. Each message is validated and stored in the local state separately, just like any other message. Thus, Pnin would store two copies of the *Challenge* message, one for each student, while the students would each store the single instance they receive.

4.3. Programming agents for enacting protocols in Argus

A conceptually simple programming model is based on the Stellar approach for Jason [26]. Here, agents react to message receptions. In Argus, we model message receptions by adding terms to the recipient's knowledge base, which they can react to using goals triggered by term addition.

Our tooling supports this programming model by generating code for the agents based on the protocol specification. Listing 6 gives the skeleton code generated for Timofey, the TA, from the grading protocol specification.

Listing 6: Generated code for TA

```
1  +rubric(MasID, Professor, TA, TID, QID, Solution)
2    : response(MasID, Student, TA, TID, QID, Question, Answer)
3    <- !send_result(MasID, TA, Professor, TID, QID, Answer, Solution).
4
5  +response(MasID, Student, TA, TID, QID, Question, Answer)
6    : rubric(MasID, Professor, TA, TID, QID, Solution)
7    <- !send_result(MasID, TA, Professor, TID, QID, Answer, Solution).
8
9  +!send_result(MasID, TA, Professor, TID, QID, Answer, Solution)
10   <- // insert code to compute result out parameters ['Grade'] here
11      .emit(result(MasID, TA, Professor, TID, QID, Answer, Solution, Grade)).
```

Our code generator focuses on enabling the emissions that the agent could send, rather than simply generating stub reactions for every reception, so the developer doesn't have to rewrite the interaction logic from the protocol. In *Grading*, TA needs to observe *rubric* and *response* before it is enabled to send *result*. These dependencies are reflected in the generated code in Listing 6 by two goals that react to the observation of *rubric* and *response* respectively. Because the messages are asynchronous and could be received in any order, both goals have guards that check for prior observation of the other message; if there are N prerequisite messages for a specific emission, the generator automatically produces N goals, each triggered by one reception and guarded by all of the others. Whichever reception is last will pass the guard, and consequently trigger the unified goal for sending the dependent message (in this case, `!send_result` for sending *result*). The developer will need to add code as indicated by the comment to bind the ⌜out⌝ parameters.

Listing 7 shows a complete implementation for Timofey, the TA, according to this model. Timofey checks if Answer equals Solution, and if so, gives a grade of 1 for that question; otherwise, 0. It then prints the grade and emits *result*.

Listing 7: TA Timofey in Argus

```
1  +rubric(MasID, Professor, TA, TID, QID, Solution)
2    : response(MasID, Student, TA, TID, QID, Question, Answer)
3    <- !send_result(MasID, TA, Professor, TID, QID, Answer, Solution).
4
5  +response(MasID, Student, TA, TID, QID, Question, Answer)
6    : rubric(MasID, Professor, TA, TID, QID, Solution)
```

```
  <- !send_result(MasID, TA, Professor, TID, QID, Answer, Solution).

+!send_result(MasID, TA, Professor, TID, QID, Answer, Solution)
  <- if (Answer = Solution) {
       .print(TID, QID, Answer, "matches", Solution);
       Grade = 1;
     } else {
       .print(TID, QID, Answer, "does not match", Solution);
       Grade = 0;
     }
     .print("Grade: ", Grade);
     .emit(result(MasID, TA, Professor, TID, QID, Answer, Solution, Grade)).
```

This programming model resembles traditional Jason communication patterns in that it is based on responding to events (which arise from incoming messages), but is enhanced with message validation based on protocols.

4.3.1. Code generation algorithms

The Argus code generator performs sophisticated analysis of protocol specifications to automatically create appropriate plan skeletons. This process relies on two key algorithms: cover calculation and conflict identification.

Emission covers For each message that an agent playing a given role can emit, the cover algorithm determines which incoming messages must be observed before the emission becomes possible. A *cover* is a set of messages that, when observed, collectively bind all the necessary ⌜in⌝ parameters required for an emission.

The algorithm first identifies all possible enactments of the protocol that include the target emission as *paths*, that is, sequences of emission and reception events. For each path, it tracks the ⌜in⌝ parameters needed by the emission and identifies observable messages that provide these parameters. When a set of messages collectively satisfies all parameter requirements, it becomes a valid cover and stops consideration of that path. Once initial covers are identified, they are pruned to eliminate redundancy. A message is removed from a cover if all the ⌜in⌝ parameters it provides are also provided by other messages in the cover. This ensures minimal covers that avoid unnecessary dependencies.

Conflict guards The conflict identification algorithm identifies messages which should block an emission when observed. These conflicts are used to add guards to the message emission plans, so they are not attempted in situations where they are blocked. The protocol adapter will enforce the constraint during emission, but the guards help avoid even attempting such violations.

For each message a role can emit, the algorithm checks all other observable messages to determine if they would disable the emission. Message A is considered to block message B if A contains parameters adorned ⌜in⌝ or ⌜out⌝ which B has adorned ⌜out⌝ or ⌜nil⌝. However, applying these criteria naively would mean that direct dependencies (with ⌜in⌝ parameters) block the message that enables them (which bind it as ⌜out⌝). While these guards would be harmless, they are unnecessary. Thus, our disablement calculation excludes messages that are the sole source of a necessary parameter.

Plan generation The code generator combines the results of these algorithms to create three types of plans:

1. For emissions with no dependencies, it generates a simple plan triggered by a goal with appropriate conflict guards. That is, the agent must set a goal to produce this message; it is not automatically triggered.
2. For emissions dependent on a single message, it generates a plan triggered by the addition of that message as a belief, which happens on reception, with conflict guards.
3. For emissions dependent on multiple messages, it generates multiple plans—one triggered by each dependency and guarded by the presence of the others, plus a goal plan to perform the actual emission.

The generation of these plans is illustrated in Listing 6, where two plans are created for the *result* message. These plans are triggered by the reception of either *rubric* or *response* and guarded by the presence of the other message, ensuring that both prerequisites are satisfied before the *result* can be emitted.

4.3.2. Agent listings

For completeness, Listings 8–10 give all the Argus agents.

Listing 8: Complete Professor Pnin in Argus.

```
question(q1, "What is your name?").
solution("lancelot", q1, "Sir Lancelot of Camelot").
solution("galahad", q1, "Sir Galahad of Camelot").
question(q2, "What is your quest?").
solution("lancelot", q2, "To seek the Holy Grail").
solution("galahad", q2, "To seek the Grail").
question(q3, "What is your favorite color?").
```

```
solution("lancelot", q3, "Blue").
solution("galahad", q3, "Yellow").

student("lancelot", "Lancelot").
student("galahad", "Galahad").

!start.

+!start <-
  TID = "midterm";
  .print("Starting test ", TID);
  .emitAll(begin_test(MasID, Professor, Student, TID));

  for (question(QID, Question)) {
    .print("Challenge ", QID, ": ", Question);
    .emitAll(challenge(MasID, Professor, Student, TID, QID, Question));
    for (student(MasID, Student)) {
      solution(MasID, QID, Solution);
      .emit(rubric(MasID, Professor, TA, TID, QID, Solution));
      .print("Solution for", MasID, QID, "is", Solution);
    };
  }.

+result(MasID, TA, Professor, TID, QID, Ans, Sol, Grade) <-
  .print("Received result for", MasID, QID, "with grade", Grade);
  .count(result(MasID, _, _, TID, _, _, _, _), C);
  .count(challenge(MasID, _, _, TID, _, _), Challenges);
  if (C >= Challenges) {
    !report(MasID, TID);
  }.

+!report(MasID, TID) : not reported(MasID, TID) <-
  .findall(Grade, result(MasID, _, _, TID, _, _, _, Grade), L);
  !sum(L, Total);
  .count(challenge(MasID, _, _, TID, _, _), C);
  .print("Total grade for student", MasID, "is", Total, "/", C);
  +reported(MasID, TID).
+!report(MasID, TID) <- true.

+!sum([], 0).
+!sum([T|R], M) <-
  !sum(R, S);
  M = T+S.
```

Listing 9: Lancelot's decision making in Argus.

```
answer("What is your name?", "Sir Lancelot of Camelot").
answer("What is your quest?", "To seek the Holy Grail").
answer("What is your favorite color?", "Blue").

+begin_test(MasID, Professor, Student, TID) <-
  .print("Starting test with TID", TID).

+challenge(MasID, Professor, Student, TID, QID, Question) : answer(Question, Answer) <-
  .print("Answering", QID, "with", Answer);
  .emit(response(MasID, Student, TA, TID, QID, Question, Answer)).
```

Listing 10: Galahad's decision making in Argus.

```
+begin_test(MasID, Professor, Student, TID) <-
  .print("Starting test with TID", TID).

+challenge(MasID, Professor, Student, TID, QID, Question) <-
  !answer(Question, Answer);
  .print("Answering", QID, "with", Answer);
  .emit(response(MasID, Student, TA, TID, QID, Question, Answer)).

```

```
+!answer("What is your name?", "Sir Galahad of Camelot").
+!answer("What is your quest?", "To seek the Grail").
+!answer("What is your favorite color?", "Blue").
+!answer(_, "I don't know that").
```

We launched all agents from a shell script. Listing 11 shows the output.

Listing 11: Example execution trace.

```
Starting Galahad...
Starting Lancelot...
Starting Timofey...
Press any key to stop...
Starting Pnin...
Pnin Starting test  midterm
Pnin Challenge  q1 :  What is your name?
Pnin Solution for lancelot q1 is Sir Lancelot of Camelot
Pnin Solution for galahad q1 is Sir Galahad of Camelot
Pnin Challenge  q2 :  What is your quest?
Pnin Solution for lancelot q2 is To seek the Holy Grail
Pnin Solution for galahad q2 is To seek the Grail
Pnin Challenge  q3 :  What is your favorite color?
Pnin Solution for lancelot q3 is Blue
Pnin Solution for galahad q3 is Yellow
Galahad Starting test with TID midterm
Lancelot Starting test with TID midterm
Galahad Answering q1 with Sir Galahad of Camelot
Lancelot Answering q1 with Sir Lancelot of Camelot
Galahad Answering q2 with To seek the Grail
Galahad Answering q3 with Blue
Lancelot Answering q2 with To seek the Holy Grail
Timofey galahad q1 Sir Galahad of Camelot matches Sir Galahad of Camelot
Lancelot Answering q3 with Blue
Timofey Grade:  1
Pnin Received result for galahad q1 with grade 1
Timofey lancelot q1 Sir Lancelot of Camelot matches Sir Lancelot of Camelot
Timofey Grade:  1
Timofey galahad q2 To seek the Grail matches To seek the Grail
Timofey Grade:  1
Timofey galahad q3 Blue does not match Yellow
Timofey Grade:  0
Timofey lancelot q2 To seek the Holy Grail matches To seek the Holy Grail
Timofey Grade:  1
Pnin Received result for lancelot q1 with grade 1
Timofey lancelot q3 Blue matches Blue
Timofey Grade:  1
Pnin Received result for galahad q2 with grade 1
Pnin Received result for galahad q3 with grade 0
Pnin Total grade for student galahad is 2 / 3
Pnin Received result for lancelot q2 with grade 1
Pnin Total grade for student lancelot is 3 / 3
Pnin Received result for lancelot q3 with grade 1
```

5. Formal semantics for Argus

Since Argus is a synthesis of protocols and BDI concepts, it helps to leverage BDI semantics. Accordingly, we begin with a brief description of the BDI semantics. Next, we introduce the key concepts of protocols and then describe the operational semantics for Argus.

5.1. Background on BDI semantics

We adopt the specific semantics for Jason to guide our choices since it is well worked out and compatible with our approach.

For the present purposes, we introduce the following elements of the Jason semantics [24, pp. 231–232]. An agent program ag comprises a set of beliefs bs and a set of plans ps. An intention is a stack of partially instantiated plans; $i[p]$ is an intention with plan p at the top. $\top$ is the empty intention. An event $e = \langle t, i \rangle$ pairs a trigger with an intention. An agent configuration $\langle ag, C, M, T, s \rangle$, includes an agent ag and four components:

- A circumstance $C = \langle I, E, A \rangle$, where (1) I is a set of intentions; (2) E is a set of events; (3) A is a set of actions.
- A messaging subsystem $M = \langle In, Out, SI \rangle$, where In is ag's inbox, Out its outbox, and SI its set of suspended intentions (as while awaiting a reply to an Ask). We don't use M or its subcomponents.
- Temporary information $T = \langle R, Ap, \iota, \epsilon, \rho \rangle$, where R is the set of relevant plans for the current event; $Ap \subseteq R$ the applicable plans (those whose contexts are true); and ι, ϵ, ρ are the intention, event, applicable plan under consideration in the current reasoning cycle.
- The current step in the agent's reasoning cycle, s. Of the nine steps defined in Jason, Argus concerns (1) ProcMsg, processing an incoming message, and (2) ExecInt, executing an intention, specifically to emit a message. In the Jason reasoning cycle (Fig. 1), ProcMsg leads to SelEv (selecting an event); ExecInt may lead to ProcMsg or to ClrInt (clearing an intention) but in our semantics leads only to ClrInt.

The Jason semantics uses the environment, env, as well, but in parts not relevant to Argus.

5.2. The Argus model

Argus adopts the circumstance, temporary information, and current step from Jason's semantics, and adds the following:

- For each pair of communicating agents, there is a point-to-point communication channel $\mathsf{Z}_{i,j}$, where $i \neq j$ are its sender and receiver, respectively. A channel's contents are an unordered set, to model unordered communication. That is, a receiver can remove a message from a channel if it is present there, independently of when that message was added relative to other messages in the channel. We can think of the channels as being part of the environment.
- We identify the local state of an agent, crucial to BSPL semantics, as $L \subseteq bs$. An agent's internal state comprises its beliefs and is also a subset of bs. However, the internal state is irrelevant to the Argus semantics, so we don't assign a symbol to it. An agent may consult (but not alter) its local state and consult (and possibly alter) its internal state, produce additional intentions, and emit and receive messages.

We adopt information protocols as introduced in BSPL [14]. A protocol is given by one or more roles and one or more message schemas involving those roles.

Definition 1 *(Message schema).* A message schema is given by $\lambda[x, y, \vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}]$, where x and y are its sender and receiver roles; $\vec{p_I}$, $\vec{p_O}$, and $\vec{p_N}$ are its (pairwise disjoint) ⌜in⌝, ⌜out⌝, and ⌜nil⌝ parameters, respectively; and $\vec{p_K} \subseteq \vec{p_I} \cup \vec{p_O}$ its key parameters.

The semantics of a protocol in BSPL is grounded in terms of the messages sent and received by the agents enacting that protocol. Specifically, although BSPL allows a protocol to refer to other protocols, for the purposes of semantics all that matters are the message schemas defined in a protocol, including the message schemas defined in the protocols referenced from it [14]. Accordingly, for a protocol P, we define $\Lambda(P)$ as the set of message schemas defined in P.

Let $\vec{p} = \vec{p_I} \cup \vec{p_O}$ be the lists of ⌜in⌝ and ⌜out⌝ parameters in a schema. We write $\overline{r}$ to indicate what a parameter is bound to and $\overline{q}$ to indicate a list of bindings corresponding to a list of parameters. (We use the same notation to avoid clutter since the context always makes clear whether we have an individual or a list.) For any parameter r, the expression $r \hookleftarrow \overline{r}$ refers to the parameter having a binding $\overline{r}$. Likewise, for any list of parameters, $\vec{q}$, the expression $\vec{q} \hookleftarrow \overline{q}$ refers to each parameter in $\vec{q}$ having the corresponding binding in $\overline{q}$.

Definition 2 *(Message instance).* A message instance $\lambda[x, y, \vec{p} \hookleftarrow \overline{p}]$ is given by a schema name λ, a sender x and a receiver y playing appropriate roles, and a payload ($\vec{p} \hookleftarrow \overline{p}$, which are bindings for ⌜in⌝ and ⌜out⌝ parameters).

5.3. Local state data model

Below, we write $\lambda[\![\ ]\!]$ to distinguish the *relation contents* from the relation (message schema) λ or $\lambda[\ ]$. We also use $\lambda[\ ]$ for message instances, since instances (comprising parameter bindings) are easily distinguished from schemas (comprising lists of key, in, out, and nil parameters).

Definition 3 *(Local state).* Let ag be an agent and let P be a protocol. Then, ag's local state (with respect to P), ag_L, is given as follows:

- For each message schema $\lambda[x, y, \vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}] \in \Lambda(P)$, where ag has adopted role x or y, ag_L contains a relation $\lambda[\![\vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}]\!]$, where this relation has a key constructed from the parameters in $\vec{p_K}$.
- ag_L contains nothing else.

A message instance maps to a ground atom, the same as a belief in Jason. That is, each message instance of a message schema λ emitted or received corresponds to a row $\lambda[\![\vec{p} \hookleftarrow \overline{p}]\!]$ being inserted in the relation for that schema. Each relation's contents at any time describe the agent's history relative to that message schema.

BSPL defines integrity independently of specific messages and considers the totality of the local states across all agents. For a *safe* protocol [14,8], if each agent separately ensures the integrity of its emissions, the multiagent system retains integrity. Local integrity means that if any two rows in any relations agree on bindings of their common key parameters, they agree on bindings of all other common parameters [27]. Accordingly, implementing the BSPL semantics requires the adapter to verify that parameter bindings present in a message instance are consistent with any bindings stored in the local state before it adds the message to its agent's local state.

The underlying intuition in BSPL is that an enactment of a protocol computes a social object. This social object is identified by the bindings of the key parameters of the protocol and contains the bindings for the other parameters of the protocol. For a parameter, in general, its binding is nonsensical unless it is associated with a key. This situation is the same as in databases. For example, in an enterprise database, *salary = 100000* is meaningless, but a row in a payroll table that captures *employeeID = ABC, salary = 100000* is indeed meaningful. That is, we cannot divorce parameters from the keys with respect to which they are defined. Recall that each message schema is an elementary protocol; thus, the foregoing applies to messages as well as to full protocols.

The above motivation leads us to impose two well-formedness criteria on protocols, motivated in Clouseau [18, §5]. First, if any two or more messages share a parameter, their keys must jointly overlap, else that parameter would have two conceptions.

Well-formedness requirement 1 *(Overlapping key parameters).* Let p be a parameter and P be a protocol. Let $\{\dots \lambda_i[x_i, y_i, \vec{p_{K,i}}, \vec{p_{I,i}}, \vec{p_{O,i}}, \vec{p_{N,i}}] \dots\}$ be the set of message schemas in P where $p \in \vec{p_{I,i}} \cup \vec{p_{O,i}} \cup \vec{p_{N,i}}$. Then, $\bigcap_i \vec{p_{K,i}} \neq \emptyset$.

That is, the set of key parameters common to these schemas is nonempty.

Second, informally, when a parameter is adorned ⌜in⌝ in a message schema, any parameter that is "essential" in identifying that parameter must also be adorned ⌜in⌝. Otherwise, that parameter's binding would be meaningless, floating in the ether as it were without being anchored to a key. The *determinant* of a parameter captures the intuition of what its essential key is. The determinant of a parameter is the intersection of the keys of all schemas in which the parameter appears.

Definition 4 *(Determinant of a parameter).* Let p be a parameter and P be a protocol. Let $\{\dots \lambda_i[x_i, y_i, \vec{p_{K,i}}, \vec{p_{I,i}}, \vec{p_{O,i}}, \vec{p_{N,i}}] \dots\}$ be the set of message schemas in P, where $p \in \vec{p_{I,i}} \cup \vec{p_{O,i}} \cup \vec{p_{N,i}}$. Then, $\Delta(p)$, the *determinant* of p is given by $\bigcap_i \vec{p_{K,i}}$.

Listing 12 illustrates the idea of a determinant through a fictitious variant of our running example. The parameter Question occurs in two message schemas, *challenge* and *responseMulti*. The parameters marked ⌜key⌝ in these message schemas are TID, QID and TID, QID, AID, respectively. Therefore, the determinant of Question in this protocol is TID, QID.

Listing 12: Determinant example

```
Determinant Example {
  roles Professor, Student, TA
  parameters out TID key, out QID key, out AID key, Question, Answer

  Professor ↦ Student: challenge[out TID key, out QID key, out Question]

  Student ↦ TA: responseMulti[in TID key, in QID key, out AID key, in Question, out Answer]
}
```

Well-formedness requirement 2 *(Parameters adorned ⌜in⌝).* Let p be a parameter and P be a protocol. Let $\lambda[x, y, \vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}]$ be a message schema in P where $p \in \vec{p_I}$. Then, $\vec{p_K} \cap \Delta(p) \subseteq \vec{p_I}$).

That is, when a parameter is adorned ⌜in⌝ in a message schema, so is each parameter in its determinant that occurs in that schema.

Notice that, in Listing 12, Question appears ⌜out⌝ in *challenge* so the above well-formed requirement is met. In addition, Question appears ⌜in⌝ in *responseMulti* as do TID and QID, so the above well-formed requirement is met there too.

5.4. Integrity checks on the local state

Given bindings for the key parameters, some parameters are deemed known, written K, if the bindings of these parameters exist in the belief base. And, given bindings for the key parameters, some parameters are deemed unknown, written U, if their bindings are not already known. An incoming message is compatible, written C, with the local state if it is consistent (with respect to its key parameters) with the bindings already present in the local state.

Below, we adopt the notation that $m \ni \vec{q} \hookleftarrow \overline{q}$ means that the specified bindings occur in the row corresponding to message instance m.

Definition 5 *(Known bindings).* The primitive $\mathsf{K}(L, \vec{k}, \vec{q} \hookleftarrow \overline{q})$ verifies if, given key parameters $\vec{k}$, the parameter bindings for $\vec{q}$ are identical to those in the local state L.

$$\mathsf{K}(L, \vec{k}, \vec{q} \hookleftarrow \overline{q}) \text{ iff } (\forall r \in \vec{q} : \ (\exists \lambda \in L, m \in \lambda[\![\ ]\!] : \ m \ni \vec{k} \hookleftarrow \overline{k} \text{ and } m \ni r \hookleftarrow \overline{r}))$$

Consider a parameter $r \in \vec{q}$. For r to be known, its determinant must be a subset of $\vec{q} \cap \vec{k}$, and its binding must be present in some message m with the same key bindings $\vec{k} \hookleftarrow \overline{k}$ in the contents of some relation $\lambda[\![\ ]\!]$.

Definition 6 *(Unknown bindings).* The primitive $\mathsf{U}(L, \vec{k} \hookleftarrow \overline{k}, \vec{q})$ verifies if, given the key bindings, the $\vec{q}$ parameters have no bindings in the local state L.

$$\mathsf{U}(L, \vec{k} \hookleftarrow \overline{k}, \vec{q}) \text{ iff } (\forall r \in \vec{q} : \ (\forall \lambda \in L : \ (\nexists m \in \lambda[\![\ ]\!] : \ m \ni \vec{k} \hookleftarrow \overline{k} \text{ and } m \ni r \hookleftarrow \overline{r})))$$

That is, for each parameter in $\vec{q}$, no message m with the same key bindings in the contents of any relation $\lambda[\![\ ]\!]$ must have a binding for that parameter.

Definition 7 *(Consistent bindings).* The primitive $\mathsf{C}(L, \vec{k}, \vec{q} \hookleftarrow \overline{q})$ verifies if the stated parameter bindings are compatible with the local state L.

$$\mathsf{C}(L, \vec{k}, \vec{q} \hookleftarrow \overline{q}) \text{ iff } (\forall r \in \vec{q}, \lambda \in L, m \in \lambda[\![\ ]\!], \overline{s} : \ \text{ if } m \ni \vec{k} \hookleftarrow \overline{k} \text{ and } m \ni r \hookleftarrow \overline{s} \text{ then } \overline{s} = \overline{r})$$

That is, for each parameter in $\vec{q}$, no message m with the same key bindings in any relation contents $\lambda[\![\ ]\!]$ has a different binding for that parameter. An absent binding is acceptable.

5.5. Operational semantics

The style of the semantic description we adopt here is called *(structural) operational semantics* [28]. Specifically, like Vieira et al. [24], our presentation matches the notion of a labeled transition system [28, p. 24] where the set of configurations is given by multiagent system configurations (Definition 11), the labels or actions are given by the actions of an agent emitting or receiving a message, and the transition relation is given by the progression of one system configuration to another (Definition 12). Our focus on message emission and reception is likewise in congruence with Plotkin's advice that "It is a matter of experience to choose the right definition of external behavior" [28, p. 20].

Our semantics shows how BDI constructs can be adapted to incorporate information protocols. Definition 8 captures an agent configuration emphasizing what is needed for communication.

Definition 8 *(Agent configuration).* The configuration of an agent ag is a tuple $\langle L, C, T, \mathsf{Step} \rangle$, where $L = ag_L$ is ag's local state, $C = ag_C$ is ag's circumstance, $T = ag_T$ is ag's temporary information, and Step is ag's current step.

Definition 9 characterizes a multiagent system and Definition 10 describes how a multiagent system enacts a protocol.

Definition 9 *(Multiagent system).* A multiagent system for P is given by $\langle Agents, \mathsf{Channels} \rangle$, where $Agents$ is a set of agents and $\mathsf{Channels}$ is a set of channels, each channel being $\mathsf{Z}_{s,r}$, where $s, r \in \mathsf{Agents}$ and $s \neq r$.

Definition 10 characterizes a MAS for a protocol: a MAS must assign agents to all of the protocol roles, and have the channels between the assigned agents that are necessary for sending all of its messages.

Definition 10 *(System for protocols).* Let P be a protocol and $MAS = \langle Agents, \mathsf{Channels} \rangle$ be a multiagent system. Let $Roles$ be the set of roles such that $x \in R$ iff x appears as sender or receiver in some schema of $\Lambda(P)$. Then, MAS is a multiagent system for P provided there is a function $Assign$, where $Assign : \ Roles \to \wp(Agents)$ is a total and injective function, and if there exists a schema in $\Lambda(P)$ with sender $Sender$ and receiver $Receiver$, then there is a corresponding channel from the agent $Assign(Sender)$ to every agent in $Assign(Receiver)$.

Definition 11 defines a configuration for a multiagent system based on the configurations of its member agents and the contents of the communication channels between them.

Definition 11 *(System configuration).* Let $MAS = \langle \{ag_0 \dots ag_n\}, \{\mathsf{Z}_{s0,r0} \dots \mathsf{Z}_{sc,rc}\} \rangle$ be a multiagent system. Then, a configuration for MAS is given by:

$$\langle \{\langle L_0, C_0, T_0, \mathsf{Step}_0 \rangle \dots \langle L_n, C_n, T_n, \mathsf{Step}_n \rangle\}, \mathsf{Z}_{s0,r0} \dots \mathsf{Z}_{sc,rc} \rangle$$

For ease of exposition, it helps to show how reasoning in Argus relates to the Jason reasoning cycle. Fig. 4 shows the Jason reasoning cycle highlighting the parts (squiggly red transitions) relevant to our semantics.

An agent's configuration may change according to the Jason reasoning cycle (the blue, solid transitions in Fig. 4). These configuration changes are not important in the Argus semantics. The remaining configuration changes (shown as squiggly red transitions in Fig. 4) correspond to the emission or reception of a message, which, in each case, also changes the requisite channel.

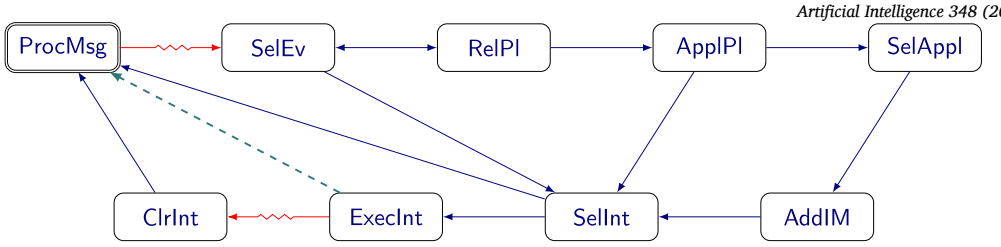


Fig. 4. Argus reasoning related to the Jason reasoning cycle. This diagram highlights the two transitions pertaining to communication, which correspond to communication-related transitions in Jason for emitting and receiving a message. The communication-related transition in Jason corresponding to a blocking emission waiting for a reply (shown dashed) is eliminated in Argus.

Definition 12 describes that a multiagent system's configuration progresses whenever an agent's configuration changes due to the emission or reception of a message, which, in each case, also changes the requisite channel to which the message is emitted or from which the message is received.

Definition 12 *(Progression).*

$$\langle\{\langle L_0, C_0, T_0, \mathsf{Step}_0\rangle \ldots \langle L_n, C_n, T_n, \mathsf{Step}_n\rangle\}, \mathsf{Z}_{s0,r0} \ldots \mathsf{Z}_{sc,rc}\rangle \longrightarrow$$

$$\langle\{\langle L'_0, C'_0, T'_0, \mathsf{Step}'_0\rangle \ldots \langle L'_n, C'_n, T'_n, \mathsf{Step}'_n\rangle\}, \mathsf{Z}'_{s0,r0} \ldots \mathsf{Z}'_{sc,rc}\rangle$$

such that there exist i and x such that $L_i \neq L'_i$ and either $\mathsf{Z}_{i,x} \neq \mathsf{Z}'_{i,x}$ or $\mathsf{Z}_{x,i} \neq \mathsf{Z}'_{x,i}$.

Notice that Definition 12 admits concurrency in that there could be two (or more) agents sending or receiving messages at the same time as long as they send and receive from channels that change. The definition prevents lockstep sending and receiving of a message. For example, if Alice sends Bob a message and Bob receives the message at the same time, Alice and Bob's local states would change but the channel from Alice to Bob that Alice sent the message to and which Bob received that message from would remain net unchanged. In BSPL, the causal intuition is crucial, meaning that sends causally precede receives, and thus do not occur concurrently. Therefore, we require a channel to first gain a message (through an emission) and only subsequently to lose that message (through a reception).

In Argus, sending is nonblocking and receptions are always enabled and nondeterministic. Point-to-point channels (Section 5.2) convey messages from the emitter to the receiver. The emitter or receiver's configuration changes in conjunction with changes to the relevant channel. Nothing else changes in the multiagent system so we leave it out for brevity.

The Argus semantics requires only two rules to describe communication, one for message emission and one for message reception, because message emission and reception are the only two actions in our model of interaction between agents. The rules below provide the elements of the structure of the transitions as characterized by Definition 12 and thus flesh out the progression of a multiagent system.

Let's briefly explain how to read each of these operational rules. On the left of the horizontal line is a name for the rule describing its purpose in our semantics. Below the horizontal line is a statement of a transition from a prior agent configuration defined abstractly (using metavariables L, C, and so on) to a posterior agent configuration defined abstractly. Above the horizontal line is a set of assertions that impose some constraints on the metavariables occurring below the line in the prior agent configuration. These assertions must hold before the rule fires for the corresponding values of the metavariables. That is, they characterize the prior agent configuration. The *where* statements below the transition define the metavariables occurring below the line in the posterior agent configuration.

5.5.1. Message emission

Given a protocol P, an agent in its ExecInt step executes an intention whose first part is to emit a message instance. We perform two tests for a message instance using its message schema and the local state: (1) the bindings of its ⌜in⌝ parameters are known in the local state, and (2) no bindings for its ⌜out⌝ and ⌜nil⌝ parameters are known in the agent's local state.

$$\textsc{Emit}\ \frac{\begin{array}{c} T_i = i[head \leftarrow .\mathsf{emit}(\lambda[ag, j, \vec{p} \hookleftarrow \overrightarrow{p}]); h] \qquad \lambda[x, y, \vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}] \in \Lambda(P) \\ \mathsf{K}(L, [\vec{p_K} \cap \vec{p_I}], [\vec{p_I} \hookleftarrow \overrightarrow{p_I}]) \qquad \mathsf{U}(L, [\vec{p_K} \cap \vec{p_I} \hookleftarrow \overrightarrow{p_K} \cap \overrightarrow{p_I}], [\vec{p_O}, \vec{p_N}]) \end{array}}{\langle L, C, T, \mathsf{ExecInt}\rangle, \mathsf{Z}_{ag,j} \longrightarrow \langle L', C', T, \mathsf{ClrInt}\rangle, \mathsf{Z}'_{ag,j}}$$

$$\textit{where}: \begin{array}{rcl} L' & = & L + \lambda[[ag, j, \vec{p} \hookleftarrow \overrightarrow{p}]] \\ C'_I & = & (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \\ Z' & = & Z \cup \{\lambda[[ag, j, \vec{p} \hookleftarrow \overrightarrow{p}]]\} \end{array}$$

Upon emitting the message, the agent advances as follows. One, it consumes its current intention, replaces it with the residual intention, and moves to the ClrInt step. Two, it updates its local state to record the bindings of the emitted message in the corresponding relation. Simultaneously with the update, it adds the emitted message to the appropriate channel.

5.5.2. Message reception

When the agent is in its ProcMsg step and a message is available in some incoming channel, it may receive the message. For the reception to proceed, we verify if the bindings are compatible with the agent's local state. Given a safe protocol and trustworthy agents, this compatibility check is redundant but we include it as good practice.

Upon receiving a message, if it passes the compatibility check, the agent advances as follows. One, it updates its local state to record the bindings of the received message in the corresponding relation. Two, it creates an event for those local state updates. Simultaneously, it removes the received message from the channel.

$$\textsc{Receive}\ \frac{\lambda[i, ag, \vec{p} \hookleftarrow \overline{p}] \in \mathsf{Z}_{i,ag} \qquad \lambda[x, y, \vec{p_K}, \vec{p_I}, \vec{p_O}, \vec{p_N}] \in \Lambda(P) \qquad \mathsf{C}(L, \vec{p_K}, \vec{p} \hookleftarrow \overline{p})}{\langle L, C, T, \mathsf{ProcMsg}\rangle, \mathsf{Z}_{i,ag} \longrightarrow \langle L', C', T, \mathsf{SelEv}\rangle, \mathsf{Z}'_{i,ag}}$$

$$\textit{where}: \quad \begin{aligned} L' &= L + \lambda[\![i, ag, \vec{p} \hookleftarrow \overline{p}]\!] \\ C'_E &= (C_E \cup \{+\langle \lambda[\![i, ag, \vec{p} \hookleftarrow \overline{p}]\!], \top\rangle\} \\ Z' &= Z \setminus \{\lambda[\![i, ag, \vec{p} \hookleftarrow \overline{p}]\!]\} \end{aligned}$$

If a message fails the compatibility check, for convenience, we "discard" it by leaving it in the channel. Because channels are unordered sets of messages, leaving a message in a channel does not block other messages in the channel from reception.

We dispense with Jason's selection (S_M) and social acceptability (SocAcc) functions—see its Tell rule (p. 244). The receiver doesn't select or reorder incoming messages. Section 6.5 describes the benefits accruing to our approach due to its avoiding the selection and social acceptability functions.

6. Contrasting Argus with communication in a plain BDI approach

Argus modifies a plain BDI approach by introducing communication protocols as explained above. Now we compare Argus with a plain BDI approach to make the case for the benefit of a careful treatment of agent communication protocols in the engineering of multiagent systems.

As explained above, we use Jason as the exemplar BDI approach with which to compare Argus. Jason's communication language is KQML [22] and Jason's operations are based on a blackboard architecture, as in tuplespaces [29]. These design choices prove to be limiting, as we explain below.

6.1. Summary of Jason's operational semantics for communications

To effectively compare Argus with a traditional BDI approach with respect to communication, it helps to bring out and explain the key part of Jason's operational semantics. Section 5.1 provides essential information to understand the rest of this subsection.

There are only two rules in the Jason semantics relevant to sending messages: one for *Ask* (and variants) and one for all the other illocutionary forces. Below is the rule called ExecActSndAsk from Vieira et al. [24, p. 241].

$$\textsc{ExecActSndAsk}\ \frac{T_i = i[head \leftarrow .\mathsf{send}(rid, ilf, cnt); h] \qquad ilf \in \{AskIf, AskAll, AskHow\}}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M', T, \mathsf{ProcMsg}\rangle}$$

$$\textit{where}: \quad \begin{aligned} M'_{Out} &= M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\ M'_{SI} &= M_{SI} \cup \{(mid, i[head \leftarrow h])\}, \\ &\quad \text{with } mid \text{ a new message identifier;} \\ C'_I &= (C_I \setminus \{T_i\}) \end{aligned}$$

In the above rule, the current intention is a plan to send a message of one of the *Ask* messages followed by a plan h to work on the response. The rule puts the message in the outbox. It removes the current intention from the circumstance but places the latter part of the suspended current intention by setting up a trigger so that when a response message arrives (of a matching *mid*) it would execute the residual plan h when that response arrives. ProcMsg is the step to process the response messages that would arrive.

Below is the rule called ExecActSnd from Vieira et al. [24, p. 242], for sending messages of other illocutionary forces besides *Ask*. This rule is simpler than the one above because it doesn't wait for a response. Therefore, the next step is not ProcMsg as above, but ClrInt.

$$\textsc{ExecActSnd}\ \frac{T_i = i[head \leftarrow .\mathsf{send}(rid, ilf, cnt); h] \qquad ilf \notin \{AskIf, AskAll, AskHow\}}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M', T, \mathsf{ClrInt}\rangle}$$

$$\textit{where}: \begin{array}{rl} M'_{Out} = & M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\ & \text{with } mid \text{ a new message identifier;} \\ C'_I = & (C_I \setminus \{T_i\}) \cup \{i[head \leftarrow h]\} \end{array}$$

Below is the rule called Tell from Vieira et al. [24, p. 244], for receiving a *Tell* message. This rule assumes a function S_M to decide which message in M_{In} to consider based on the priority assigned by a programmer. The rule also assumes a function SocAcc, which applies social reasoning to determine which action to take given the "social acceptability" of the message.

$$\textsc{Tell}\ \frac{\begin{array}{c} S_M(M_{In}) = \langle mid, id, Tell, Bs\rangle \\ (mid, i) \notin M_{SI} \text{ for any intention } i\mathsf{SocAcc}(id, Tell, Bs) \end{array}}{\langle ag, C, M, T, \mathsf{ProcMsg}\rangle \longrightarrow \langle ag, C', M', T, \mathsf{SelEv}\rangle}$$

$$\textit{where}: \begin{array}{rl} M'_{In} = & M_{In} \setminus \{\langle mid, id, Tell, Bs\rangle\} \\ \text{for each } b \in Bs: & \\ ag'_{bs} = & ag_{bs} + b[id] \\ C'_E = & C_E \cup \{\langle +b[id], \top\rangle\} \end{array}$$

Below is Jason's MsgExchg rule (p. 242) that describes how the configuration of a multiagent system progresses given how the configurations of the agents in the system progress when one of the agents sends a message to another agent.

$$\textsc{MsgExchg}\ \frac{\langle mid, id_j, ilf, cnt\rangle \in M_{i,Out}}{\{ag_{id_1} \ldots ag_i \ldots ag_{id_j} \ldots ag_{id_n}, env\} \longrightarrow \{ag_{id_1} \ldots ag'_i \ldots ag'_{id_j} \ldots ag_{id_n}, env\}}$$

$$\textit{where}: \begin{array}{rl} M'_{i,Out} = & M_{i,Out} \setminus \{\langle mid, id_j, ilf, cnt\rangle \in M_{i,Out}\} \\ M'_{j,In} = & M_{j,In} \cup \{\langle mid, id_i, ilf, cnt\rangle \in M_{j,In}\} \end{array}$$

Below is Jason's NoMsg rule (p. 249) that describes that an agent may proceed from its ProcMsg step when no message is present in its inbox.

$$\textsc{NoMsg}\ \frac{M_{In} = \{\ \}}{\langle ag, C, M, T, \mathsf{ProcMsg}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{SelEv}\rangle}$$

6.2. Fixed message types and meanings

KQML features a predetermined set of message types. A fixed set of message types (and associated meanings) is unnecessarily limiting because meaning arises from the application domain. There is no way that the designers of KQML (or any language) could anticipate all possible meanings [30, Singh's essay, pp. 15–16]. For example, a price supplied for an item may have these meanings: (1) offer to sell at that price, as in e-commerce; (2) report of a recent transaction, as in stock markets; (3) current bid; (4) predicted price for a security, as in an analyst's report. In practice, therefore, programmers simply use the *Tell* primitive as a transport wrapper and hope the meaning is clear from the payload. That is, the message type carries no practical information. Hence, there is no support for programmers to express the meanings of relevance to the application.

To get a sense of the semantic variety of speech acts, consider that Vanderveken [31, ch. 6] discusses 70 English assertives, 32 commissives, 56 directives, 85 declaratives, and 28 expressives. KQML, by contrast, features only about two dozen speech acts, all assertives or directives, and several of which are merely operational variants. Jason features a strict subset of KQML's speech acts (Table 1).

Argus dispenses with KQML primitives, such as *Ask* and *Tell*, which are central to Jason's communication model. Instead, Argus simply provides a generic primitive to emit whatever application-specific messages are specified in the protocol. A message can be understood in illocutionary [23,32] terms as follows. The ⌜in⌝ parameters in a message together with the ⌜key⌝ parameters can be understood as forming an informative and the ⌜out⌝ parameters together with the ⌜key⌝ parameters can be understood as forming a declarative.

In general, the application meaning of a message can be made explicit via representations such as those based on commitments and other norms [33,34]. For example, an offer message may be modeled as creating a practical commitment and a prediction message may be modeled as creating a dialectical commitment [35]. Although such representations are out of scope for this paper, application-specific communications provide the basis for capturing them, in contrast to KQML primitives such as *Tell*.

An important point is that the BSPL semantics provides precise computational interpretations for the parameter adornments (⌜in⌝, ⌜out⌝, and ⌜nil⌝). These interpretations are supported and enforced by the Argus architecture and are not left up to the decision

making of an agent. In contrast, BSPL has nothing to say about commitments or other norms. Thus, how agents process the meanings of the messages they exchange is captured in some other way, e.g., in higher-level protocols, which are not within the scope of the present paper.

6.3. Request-response communication

When a Jason agent sends some variant of the *Ask* message type using the ExecActSndAsk rule, the Jason reasoning cycle transitions from ExecInt (where the message emission occurs) to ProcMsg (where the agent processes an incoming message, in this case, the reply) before continuing with that intention. That is, an intention is suspended upon sending the message until a matching reply arrives. In essence, this represents a BDI encoding of remote-procedure calls, a pattern from client-server computing that is well-known to be inapplicable where autonomous agents interact.

Argus does not use the ExecActSndAsk rule, instead replacing all message emissions with the single Emit rule, which is asynchronous—i.e., it does not wait for a reply. Thus, the Argus reasoning cycle always goes from ExecInt to ClrInt during emission.

Clearly, there could be cases where an agent must wait until it receives some information: protocols provide a simple way to express such dependencies and Argus supports enacting them. The design limitation is that Jason enshrines the two-party request-response pattern in its semantics whereas real-life communications are not always so constrained. For example, three-party interactions are common (as Listing 4 shows, in our running example, Professor sends a message to Student, who sends a response to TA). Jason's selection of the two-party request-response pattern as special may have been motivated by adherence to KQML rather than any fundamental principle.

Fig. 4 shows the reasoning cycle for an Argus agent, which is identical to Jason's reasoning cycle except for the removal of the path between ExecInt and ProcMsg (which is still drawn, but as a dashed line to highlight which transition was removed).

6.4. Correlation by syntactic identifiers

Argus' programming model takes advantage of keys in BSPL to correlate information, as Listing 7 illustrates. The order in which the *rubric* and *response* for a given question are received is unimportant because they are guarded and matched based on the enactment keys, TID and QID. Crucially, the keys come from the application domain and, therefore, are semantic in nature. From the point of view of implementation, Argus benefits from the logic programming model underlying Jason, which can accommodate semantic identifiers.

By contrast, Jason's communication model lacks semantic identifiers. This means that identifiers and correlation must be implemented in agents in ad hoc ways, leading to tight coupling between agents [15]. An identifier *mid*, as used in ExecActSndAsk and other rules, is defined as a message identifier but is used as a conversation identifier, i.e., for correlation, since it is reused by a response message. Such identifiers are "syntactic" because they are unrelated to the application domain.

Chopra and Singh [36] classify the use of such identifiers as a semantic antipattern because by artificially compartmentalizing elements of the application state, such identifiers block necessary semantic inference. Drawing upon Chopra and Singh's example, commitment reasoning, which is at a higher level than Jason's conversations, would not work on a store of messages in Jason. By contrast, commitment reasoning works naturally on top of the local state [18,15]. One might insert semantic keys in the payload of a Jason (i.e., KQML) message. Whereas, in general, inserting semantic keys would be an improvement, it would mean we would have two standards of keys and still no support from the architecture.

Further, a single identifier such as *mid* is inadequate for modeling subconversations [15]. In terms of our running example, every challenge in a test may be thought of as a subconversation.

6.5. Hidden dependencies via external functions

Jason's selection S_M and social acceptability SocAcc are *hidden* functions in the sense that neither S_M nor SocAcc is based on the state of the receiving agent, as captured by its beliefs. S_M is a function of the inbox of the agent. Thus, it presumably has access to the messages in the inbox. Each incoming message is given by a four-tuple $\langle mid, id, ilf, cnt \rangle$, where *mid* is a message identifier, *id* identifies the sending agent, *ilf* is the illocutionary force of the message (e.g., *Tell*), and *cnt* is its content (such as assertions being) [24, p. 241]. SocAcc is a function of the body of a message, i.e., everything except its message identifier, *mid*. Since they are not based on the receiving agent's state, it's unclear how these functions can reasonably be computed. It appears they are hardcoded, if used at all.

S_M and SocAcc go against the BDI paradigm. Relying on them means that the receiver's beliefs do not reflect its information about the world because some of the state is hidden in the outcomes of these functions. In essence, the receiving agent's behavior is not characterized by its beliefs and intentions. S_M and SocAcc also go against interaction orientation by capturing part of the interaction while being internal to an agent. These functions thus form hidden dependencies between agents by coupling their internals so that the multiagent system progresses as desired.

In general, as a physical system, the agent knows [37] that a message has arrived because the agent has performed computations based on it. Omitting the knowledge of a message from the agent's beliefs is not only an exercise in self-deception but also enables antipatterns in multiagent architecture [15]. In addition, external functions such as S_M and SocAcc exacerbate complexity for the

programmer since the agent's control state is split between these functions and the agent's intentions. The two representations must be constantly reconciled.

An Argus agent also processes messages available for reception—as encoded in the RECEIVE rule—before making them part of the agent's belief base (specifically, its local state). However, whereas Jason's S_M and SocAcc are intended to capture an agent's decision making, the processing in the RECEIVE encodes protocol semantics, not an agent's decision making.

6.6. Tackling prioritization and social acceptability in Argus

We emphasize that prioritization and social acceptability are valuable capabilities for an agent. For example, a merchant may prioritize shipping goods based on its inventory and may offer discounts or credits to customers of long standing. With Argus, priorities and social relationships are readily captured using beliefs and plans, as Listing 13 demonstrates. Notably, Vieira et al. [24][p. 244] indeed note the possibility of capturing social constraints using practical reasoning as an alternative to SocAcc.

Listing 13: Prioritizing TA in Argus.

```
student("s1", 3). // ID, year
student("s2", 4).

reverse([], Z, Z) :- true.
reverse([H|T], Z, Acc) :- reverse(T, Z, [H|Acc]).

+response(MasID, Student, TA, TID, QID, Question, Answer)
  : rubric(MasID, Professor, TA, TID, QID, Solution) <-
  +task(MasID, TID, QID, Answer, Solution).

+rubric(MasID, Professor, TA, TID, QID, Solution)
  : response(MasID, Student, TA, TID, QID, Question, Answer) <-
  +task(MasID, TID, QID, Answer, Solution).

+task(MasID, TID, QID, Answer, Solution)
  : .count(task(_, _, _, _, _), C) & C = 6
  <- !prioritize(P);
  !work(P).

+!work(P) <-
  for(.member([Year, TID, QID], P)) {
    task(MasID, TID, QID, Ans, Sol);
    !grade(MasID, TID, QID, Ans, Sol);
  }.

+!map_year([[TID, QID] | []], P) : student(TID, Year) <-
  P = [[Year, TID, QID]].
+!map_year([[TID, QID]|T], P) : student(TID, Year) <-
  !map_year(T, P2);
  P = [[Year, TID, QID] | P2].

+!prioritize(P) <-
  .findall([TID, QID], task(MasID, TID, QID, Ans, Sol), L);
  !map_year(L, L2);
  .sort(L2, L3);
  reverse(L3, P, []).

+!grade(MasID, TID, QID, Answer, Solution) <-
  if (Answer == Solution) {
    .print(TID, QID, Answer, "matches", Solution);
    Grade = 1;
  } else {
    .print(TID, QID, Answer, "does not match", Solution);
    Grade = 0;
  }
  .print("Grade: ", Grade);
  .emit(result(MasID, TA, Professor, TID, QID, Answer, Solution, Grade)).
```

In Listing 13, the TA does not directly initiate a goal for grading results but instead creates tasks that can be prioritized. Timofey waits until he has six available tasks before starting his work (elided in the abridged version), so he can prioritize them using social constraints, namely student seniority. Timofey prioritizes senior students (who may need to graduate or register for classes earlier) by sorting the tasks according to the students' years in college.

6.7. Synchronization via shared artifacts

Jason adopts the idea of a blackboard architecture for coordinating cooperative problem solvers [38,39]. Jason's MsgExchg rule (p. 242) assumes synchronization since a message appears in the sender and receiver's configurations simultaneously.

Carriero and Gelernter [29,40] describe *tuplespaces*, an approach for coordinating distributed computation where agents read from and write on a shared space (i.e., a blackboard) to exchange information. CArtAgO [41], which envisages Jason agents interacting through shared artifacts in the environment, supports tuplespaces as a particular type of coordination artifact.

The main idea behind tuplespaces is separating coordination from computation—a major step forward at the time. Common operations on tuplespaces include *in(t)*, which blocks until it can take a matching tuple (removing it from the tuplespace), *rd* which is like *in* but copies the tuple instead of removing it, and *out*, which writes a tuple to the tuplespace. These operators create information dependencies similar to the ⌜in⌝ and ⌜out⌝ parameter adornments of BSPL: an *in* cannot return until the corresponding *out* completes. Although similar, there are significant conceptual differences, most crucially that tuplespaces leave the relationships between operations on tuples up to the agent implementations. Thus, an agent programmer must implement the correct sequence of tuplespace operations without programming model support. Further, the programmers of different agents must agree on their implementations for purposes of interoperability, thus compromising loose coupling. BSPL avoids these problems by explicitly modeling parameter relationships in the message schemas in protocols, which Argus exploits in a programming model that supports implementing agents.

Also, tuplespaces require an agent to select information to read for processing. As Section 6.5 explains, this approach places decision-making outside of the cognitive model of the agent.

Our semantics incorporates asynchrony in a shared-nothing setting (avoiding synchronization primitives) [42] that enables agents to proceed in a maximally decoupled manner, constrained only by the causal dependencies between the messages they send one another.

We note that some implementations of tuplespaces support nonblocking versions of *in* and *rd*; regardless, the points we make above remain valid.

6.8. Comparing the programming models

Jason doesn't support communication protocols. Vieira et al. [24] state that "communication structures" can be captured as plans (p. 253). However, plans are placed within agents. It is well known that plans are not protocols, which are about interaction [3]. Moreover, adding protocols to Jason is nontrivial because Jason's design suffers from crucial limitations, which we described above.

The use of protocols at the heart of a programming model in Argus facilitates the implementation of agents by encoding the protocol-specific reasoning in the generated code and the protocol adapter, enabling programmers to focus on the agent's decision making. The generated code provides clear points for plugging in the decision making and guides the implementation of agents so that they attempt to emit only legal (enabled) messages. The agent's adapter enforces all protocol constraints, ensuring that its emissions and receptions accord with its local state. The adapter is crucial for catching errors in agent implementations. In this manner, the generated code and the adapter support correctness as well.

Argus supports interoperability and loose coupling. Therefore, beyond agreeing on the protocol and developing agents to enact the respective roles in the protocol, the programmers do not need to coordinate their design choices. The agents may be contributed by multiple programmers in different organizations.

There is a small caveat though. The Argus adapter sends and received JSON-encoded messages over UDP and materializes them as beliefs. As long as agents use the Argus adapter, interoperability is guaranteed. However, if agents want to use different BSPL adapters, then standardizing on the message encoding and transport becomes necessary. Such standardization would indeed be valuable.

In Argus, messages arrive nondeterministically, including when a response is expected to a previous query. Moreover, Argus doesn't have any use for suspended intentions to deal with requests in progress because the local state is captured entirely in the parameter bindings. The internals of an agent can function arbitrarily.

7. Case study: HL7

We highlight the benefits of Argus by modeling a healthcare scenario specified by HL7, a health standards body, as a UML activity diagram [43] (reproduced as Fig. 9 in the Appendix). The scenario describes the process for creating a lab order—that is, for a health care provider to request collection of (or collect themselves) a sample from the patient and then perform lab work on the sample. The scenario is flexible because there are three ways that samples could be collected. Thus, there could be multiple ways to report that work is completed depending on the needs of the agents. Our model involves four agents: Patient, Provider, Collector, and Laboratory. We elide the interactions between these roles and their computer systems (which the original UML includes) because we focus on interactions between autonomous parties. For the present study, we modeled the scenario as an information protocol based on the formalization of Christie V et al. [44] with some corrections and simplifications. Listing 14 shows this protocol.

The main interaction is the CreateOrder protocol, which begins with Patient submitting a complaint to Provider, who then communicates the need for lab work to Laboratory. The interaction then branches into three possible protocols for sample collection: (1) the provider can collect the sample themselves, (2) the provider can instruct the subject to visit the lab, who performs the collection, (3) or the provider can instruct the subject to get an appointment with a third party specialist who performs the collection.

Once the lab work has been performed, the lab communicates the results to the provider, either directly or by notifying them that they are available for querying.

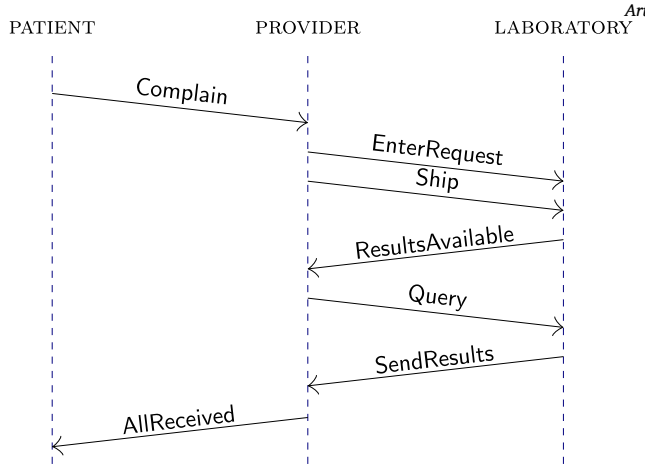


Fig. 5. Possible Provider Collection enactment, with querying for results.

We have written a protocol adapting the workflow, focusing on the communication and flow of information. Some of the other actions, such as transporting samples, are also adapted as messages; these could be interpreted as notifications about the sample transfer instead of sending digital samples only.

Listing 14: The main interaction flow of the HL7 Create Laboratory Order standard expressed in BSPL.

```
CreateOrder {
  roles Patient, Provider, Collector, Laboratory
  parameters out ID key, out complaint, out report
  private order, collection, specimen, unsuitable, received, results, results-id, co-location, contact,
    query, request, order-query, order-response

  Patient ↦ Provider: Complain[out ID, out complaint]
  Provider ↦ Laboratory: EnterRequest[in ID, in complaint, out order]

  Provider ↦ Laboratory: Ship[in ID, in order, out collection, out specimen]
  Provider ↦ Collector: NonProviderCollect[in ID, in order, out collection, out co-location]
  Provider ↦ Patient: NeedAppointment[in ID, in order, out collection, out contact]

  Patient ↦ Collector: Schedule[in ID, in contact, out co-location, nil specimen]
  Collector ↦ Provider: OrderQuery[in ID, in co-location, nil order, out order-query]
  Provider ↦ Collector: OrderResponse[in ID, in order-query, in order, out order-response]
  Collector ↦ Laboratory: CollectSpecimen[in ID, in order, in co-location, out specimen]

  Laboratory ↦ Collector: NotifyUnsuitable[in ID, in order, in specimen, out unsuitable]
  Laboratory ↦ Collector: NotifyReceived[in ID, in order, in specimen, out received]

  Laboratory ↦ Provider: Results[in ID, in order, in specimen, out results]
  Laboratory ↦ Provider: ResultsAvailable[in ID, in order, in specimen, out results-id]

  Provider ↦ Laboratory: Query[in ID, in results-id, out query]
  Laboratory ↦ Provider: SendResults[in ID, in results-id, in query, out results]

  Provider ↦ Patient: AllReceived[in ID, in results, out report]
}
```

The above listing gives the complete specification of the interaction. In this collection of actions, PATIENT sends a complaint to PROVIDER. Based on this complaint, PROVIDER creates an order for laboratory work and sends it to LABORATORY. Then, one of the three methods for sample collection are performed: the provider can collect the specimen, or they can delegate collection to COLLECTOR, to be performed either immediately or at a later scheduled appointment. After the collection has been performed, LABORATORY may respond in several ways, notifying that they've received the samples, possibly flagging them as unsuitable and requiring another collection, and returning results either directly or via notification for retrieval. When all the work has been completed, PROVIDER sends the final results to PATIENT. Figs. 5–7 demonstrate protocol enactments corresponding to the three collection methods.

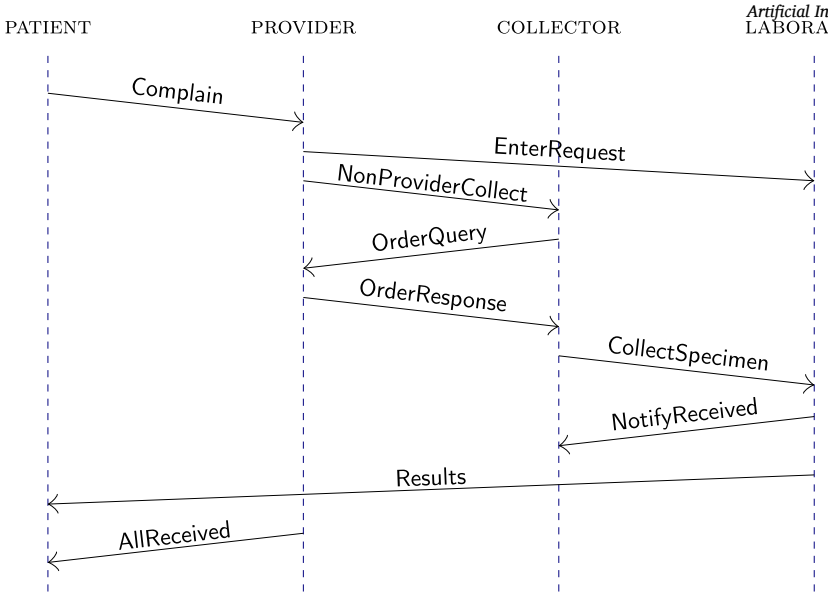


Fig. 6. Possible Nonprovider Collection enactment.

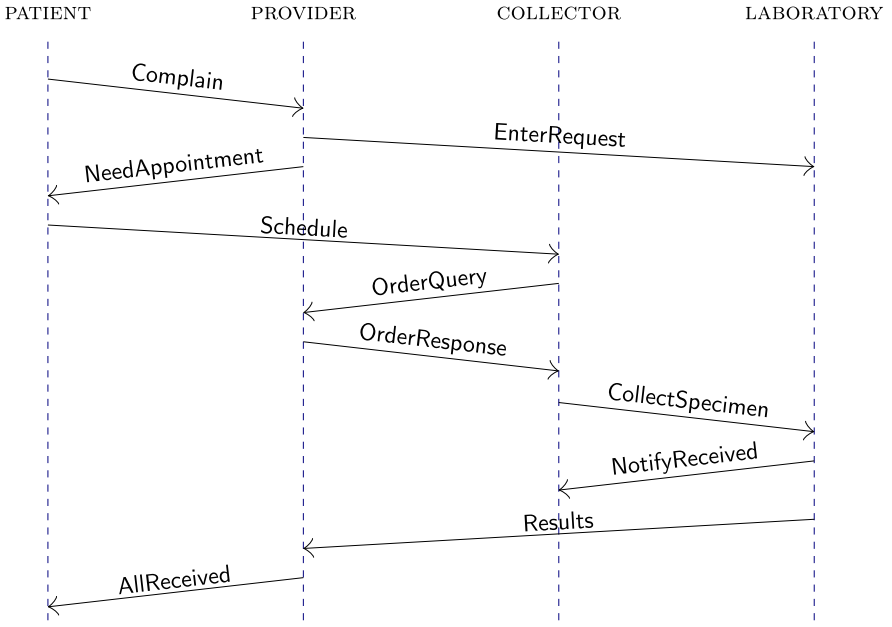


Fig. 7. Possible Scheduled Collection enactment.

7.1. Role skeletons

Our tooling generates the agent skeletons from the protocol. Specifically, Listings 15 and 16 respectively state the skeletons for roles PATIENT and PROVIDER. The skeletons clearly indicate where programmers need to plug in internal reasoning. In the absence of the protocol, as is currently the case with Jason, a programmer would have no basis for structuring agents as such and would have to figure out the entire agent design from scratch. (Generated skeletons for all roles are available in the online code repository. In the light of Jason's syntactic conventions, in the generated code, Uppercase and CamelCase message names are automatically converted to lowercase names with '_' between component names and lowercase parameter names are automatically converted into Uppercase names.)

Listing 15: Agent skeletons for role PATIENT automatically generated from the protocol of Listing 14.

```
+!send_complain
  <- // insert code to compute Complain out parameters ['ID', 'complaint'] here
     .emit(complain(MasID, Patient, Provider, ID, Complaint)).

+need_appointment(MasID, Provider, Patient, ID, Order, Collection, Contact)
  <- // insert code to compute Schedule out parameters ['co-location'] here
     .emit(schedule(MasID, Patient, Collector, ID, Contact, CoLocation)).
```

Listing 16: Agent skeleton for role PROVIDER automatically generated from the protocol of Listing 14.

```
+complain(MasID, Patient, Provider, ID, Complaint)
  <- // insert code to compute EnterRequest out parameters ['order'] here
     .emit(enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)).

+enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  : not non_provider_collect(MasID, Provider, Collector, ID, Order, Collection, CoLocation) & not
     need_appointment(MasID, Provider, Patient, ID, Order, Collection, Contact) & not results(MasID,
     Laboratory, Provider, ID, Order, Specimen, Results) & not results_available(MasID, Laboratory,
     Provider, ID, Order, Specimen, ResultsId)
  <- // insert code to compute Ship out parameters ['collection', 'specimen'] here
     .emit(ship(MasID, Provider, Laboratory, ID, Order, Collection, Specimen)).

+enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  : not ship(MasID, Provider, Laboratory, ID, Order, Collection, Specimen) & not need_appointment(MasID
     , Provider, Patient, ID, Order, Collection, Contact) & not order_query(MasID, Collector, Provider,
     ID, CoLocation, OrderQuery)
  <- // insert code to compute NonProviderCollect out parameters ['co-location', 'collection'] here
     .emit(non_provider_collect(MasID, Provider, Collector, ID, Order, Collection, CoLocation)).

+enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  : not ship(MasID, Provider, Laboratory, ID, Order, Collection, Specimen) & not non_provider_collect(
     MasID, Provider, Collector, ID, Order, Collection, CoLocation)
  <- // insert code to compute NeedAppointment out parameters ['collection', 'contact'] here
     .emit(need_appointment(MasID, Provider, Patient, ID, Order, Collection, Contact)).

+order_query(MasID, Collector, Provider, ID, CoLocation, OrderQuery)
  : enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  <- !send_order_response(MasID, Provider, Collector, ID, OrderQuery, Order).

+enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  : order_query(MasID, Collector, Provider, ID, CoLocation, OrderQuery)
  <- !send_order_response(MasID, Provider, Collector, ID, OrderQuery, Order).

+!send_order_response(MasID, Provider, Collector, ID, OrderQuery, Order)
  <- // insert code to compute OrderResponse out parameters ['order-response'] here
     .emit(order_response(MasID, Provider, Collector, ID, OrderQuery, Order, OrderResponse)).

+results_available(MasID, Laboratory, Provider, ID, Order, Specimen, ResultsId)
  <- // insert code to compute Query out parameters ['query'] here
     .emit(query(MasID, Provider, Laboratory, ID, ResultsId, Query)).

+results(MasID, Laboratory, Provider, ID, Order, Specimen, Results)
  <- // insert code to compute AllReceived out parameters ['report'] here
     .emit(all_received(MasID, Provider, Patient, ID, Results, Report)).

+send_results(MasID, Laboratory, Provider, ID, ResultsId, Query, Results)
  <- // insert code to compute AllReceived out parameters ['report'] here
     .emit(all_received(MasID, Provider, Patient, ID, Results, Report)).
```

7.2. Agents

Patient-1 is an agent that implements role PATIENT by fleshing out the skeleton in Listing 15.

Listing 17: Patient-1, an agent playing role PATIENT.

```
!send_complain.
+!send_complain
  <- MAS = "main";
     ID = 1;
     Complaint = "My toe hurts.";
     .print("Complaining: ", Complaint);
     .emit(complain(MAS, Patient, Provider, ID, Complaint)).

+need_appointment(MasID, Provider, Patient, ID, Order, Collection, Contact)
  <- CoLocation = "lab 4";
     .emit(schedule(MasID, Patient, Collector, ID, Contact, CoLocation, Specimen)).

+all_received(MasID, Provider, Patient, ID, Results, Report)
  <- .print(Report).
```

As shown above, completing the agent specification is as simple as binding the identified out parameters. This can be as simple as setting them to a constant, or could involve more complex logic that selects the values based on the bindings of other parameters. The Argus protocol adapter automatically manages the beliefs corresponding to message observations, triggering these goals when their message is received. The skeleton generator applies heuristics to decide when it is likely or possible to emit the messages, as shown above.

Listing 18: Provider-1, an agent playing the PROVIDER role.

```
+complain(MasID, Patient, Provider, ID, Complaint)
  <- Order = "Order 0001";
     .emit(enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)).

+enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  : not ship(MasID, Provider, Laboratory, ID, Order, Collection, Specimen) & not need_appointment(MasID
     , Provider, Patient, ID, Order, Collection, Contact) & not order_query(MasID, Collector, Provider,
     ID, CoLocation, OrderQuery)
  <- .randint(1, 3, Method);
     if (Method == 1) {
       Collection = "provider";
       Specimen = "Specimen 0000";
       .emit(ship(MasID, Provider, Laboratory, ID, Order, Collection, Specimen));
     };
     if (Method == 2) {
       Collection = "non-provider";
       CoLocation = "on-site";
       .emit(non_provider_collect(MasID, Provider, Collector, ID, Order, Collection, CoLocation));
     };
     if (Method == 3) {
       Collection = "appointment";
       Contact = "Laboratory";
       .emit(need_appointment(MasID, Provider, Patient, ID, Order, Collection, Contact));
     }.

+order_query(MasID, Collector, Provider, ID, CoLocation, OrderQuery)
  : enter_request(MasID, Provider, Laboratory, ID, Complaint, Order)
  <- OrderResponse = "responded";
     .emit(order_response(MasID, Provider, Collector, ID, OrderQuery, Order, OrderResponse)).

+results_available(MasID, Laboratory, Provider, ID, Order, Specimen, ResultsId)
  <- Query = "get results";
     .emit(query(MasID, Provider, Laboratory, ID, ResultsId, Query)).

+results(MasID, Laboratory, Provider, ID, Order, Specimen, Results)
  <- !report(MasID, Provider, Patient, ID, Results).
+send_results(MasID, Laboratory, Provider, ID, ResultsId, Query, Results)
  <- !report(MasID, Provider, Patient, ID, Results).

+!report(MasID, Provider, Patient, ID, Results)
  <- Report = "Negative";
     .emit(all_received(MasID, Provider, Patient, ID, Results, Report)).
```

The completed implementation (Provider-1) in Listing 18 shows several common patterns. First, multiple branches are folded into one goal, selected by appropriate conditional logic. Second, some of the branches that result in emitting the same message are refactored to trigger a common goal that handles that emission, reducing redundancy.

7.3. Execution trace

Listing 19 shows an enactment generated upon launching the implemented agents in a shell.

Listing 19: Enactment Trace

```
Patient-1 Complaining:  My toe hurts.
2024-06-18 17:01:25,365 (Patient-1): Sent: Complain(ID=1.0, complaint='My toe hurts.'){system='main'}
2024-06-18 17:01:25,366 (Provider-1): Received: Complain(ID=1.0, complaint='My toe hurts.'){system='main', received=datetime.datetime(2024, 6, 18, 17, 1, 25, 366037)}
2024-06-18 17:01:25,367 (Provider-1): Sent: EnterRequest(ID=1.0, complaint='My toe hurts.', order='Order 0001'){system='main'}
2024-06-18 17:01:25,368 (Laboratory-1): Received: EnterRequest(ID=1.0, complaint='My toe hurts.', order='Order 0001'){system='main', received=datetime.datetime(2024, 6, 18, 17, 1, 25, 368093)}
2024-06-18 17:01:25,368 (Provider-1): Sent: Ship(ID=1.0, order='Order 0001', collection='provider', specimen='Specimen 0000'){system='main'}
2024-06-18 17:01:25,369 (Laboratory-1): Received: Ship(ID=1.0, order='Order 0001', collection='provider', specimen='Specimen 0000'){system='main', received=datetime.datetime
(2024, 6, 18, 17, 1, 25, 369573)}
Laboratory-1 4
2024-06-18 17:01:25,370 (Laboratory-1): Sent: NotifyReceived(ID=1.0, order='Order 0001', specimen='Specimen 0000', received='received'){system='main'}
2024-06-18 17:01:25,371 (Laboratory-1): Sent: Results(ID=1.0, order='Order 0001', specimen='Specimen 0000', results='negative'){system='main'}
2024-06-18 17:01:25,371 (Collector-1): Received: NotifyReceived(ID=1.0, order='Order 0001', specimen='Specimen 0000', received='received'){system='main', received=datetime.datetime
(2024, 6, 18, 17, 1, 25, 371099)}
2024-06-18 17:01:25,371 (Provider-1): Received: Results(ID=1.0, order='Order 0001', specimen='Specimen 0000', results='negative'){system='main', received=datetime.datetime
(2024, 6, 18, 17, 1, 25, 371511)}
2024-06-18 17:01:25,372 (Provider-1): Sent: AllReceived(ID=1.0, results='negative', report='Negative'){system='main'}
2024-06-18 17:01:25,372 (Patient-1): Received: AllReceived(ID=1.0, results='negative', report='Negative'){system='main',
received=datetime.datetime(2024, 6, 18, 17, 1, 25, 372754)}
Patient-1 Negative
```

7.4. Pure Jason implementation

In contrast with the above automatically-generated skeletons and their implementations, we have also implemented a Jason version of the MAS directly from HL7 specifications, that is, without the benefit of the Argus protocol-based approach.

Listing 20: Agent Patient-2 in Jason that captures same functionality as the Argus Patient-1 (Listing 17).

```
complaint(1, "stomach pain").
complaint(2, "insomnia").
complaint(3, "fever").
complaint(4, "skin lesion").

!register.
+!register <- .df_register(patient).

!start.

+!start <-
  .df_search(provider, Provider);
  for(complaint(ID, Symptom)) {
    // 1.1 Patient Presents with Complaint
    .send(Provider, tell, complaint(ID, Patient, Symptom));
    .print("Complained to ", Provider, ": ", Symptom);
  }.

```

```
+complete(ID) <-
  .print("All orders complete for complaint ", ID).

// 4.1 Patient Calls for Spec Collect Appt.
+schedule_appointment(ID) <-
  .df_search(phlebotomist, Ph);
  .my_name(Patient)
  .send(Ph, achieve, collect(ID, Patient));
  .print("Called phlebotomist for appointment about ", ID).
```

Listing 20 gives the agent code for role PATIENT in *CreateOrder*, including simplified internal logic and decision-making. The patient searches for an agent playing PROVIDER, and sends their complaints (multiple, to simulate several interactions) to that agent. The patient also listens for a complete event and has a goal for scheduling appointments when necessary.

Among the differences between the Jason implementation and the Argus version given in Listing 17 is the way that the recipients are identified and handled. Where Jason relies extensively on the Directory Facilitator (df) to register and search for agents, the Argus adapter explicitly tracks roles and their bindings under a MAS, identified via MasID in these listings. Thus, the Jason program constantly has to look up the next recipient, which may become problematic if there is more than one agent available to play each role—there is no guarantee that all of the messages from the same enactment will go to the same agent.

Listing 21: Agent Provider-2 in Jason.

```
!register.
+!register <- .df_register(provider).

// 2.1 Provider Evaluates Complaint
+complaint(ID, Patient, Symptom)[source(Patient)] <-
  .random(R);
  // 2.2 Lab Diagnostic Testing Needed?
  if (R > 0.1) { // 90 percent chance of deciding to run lab test
    // 3.1 Provider Enters Request into POS
    +request_test(ID, Patient, Symptom);
  } else {
    .print("No test needed for ", Symptom);
  }.

+request_test(ID, Patient, Symptom) <-
  .print("Requesting test for ", Symptom);
  .random(Who);
  if (Who <= 0.3) {
    !provider_collect(ID, Patient, Symptom);
  } elif (0.3 <= Who & Who <= 0.6) {
    !appointment(ID, Patient);
  } else {
    !non_provider_collect(ID, Patient);
  }.

// 2.2 [sic] Provider Collects Specimen
+!provider_collect(ID, Patient, Symptom) <-
  .print("Provider collecting sample for case ", ID);
  .df_search(lab, Laboratory);

  // 2.3 Specimen Labeled and Shipped to Laboratory
  .send(Laboratory, tell, specimen(ID, "specimen"));
  // 3.4 Push Electronic Connectivity to Lab?
  // 3.5 Send Laboratory Order to Laboratory
  .send(Laboratory, tell, lab_order(ID, Patient, Symptom));
  .print("Sent lab_order ", ID).

+!appointment(ID, Patient) <-
  .print("Scheduling appointment for patient ", ID);
  .send(Patient, tell, schedule_appointment(ID)).

+!non_provider_collect(ID, Patient) <-
  .df_search(phlebotomist, Ph);
  .print("Directing phlebotomist to collect specimen from patient ", ID);
  .send(Ph, achieve, collect(ID, Patient)).

```

Table 3
Bugs found during code development.

Programming Error	Count
Naming inconsistencies	4
Incorrect parameter ordering or count	4
Incorrect syntax for adding beliefs	1

```
47 // 3.6 Respond to Lab Order Query
48 +!send_lab_order(ID)[source(Requestor)] : complaint(ID, Patient, Symptom) <-
49   .send(Requestor, tell, lab_order(ID, Patient, Symptom));
50   .print("Responded to Lab Order Query ", ID).
51
52 +results_available(ID)[source(Laboratory)] <-
53   // 2.5 Receive notification
54   .print("Received notification that results are available for order ", ID);
55   // 2.6 Query Results
56   .send(Laboratory, achieve, query_results(ID));
57   .print("Queried results for ", ID).
58
59 +results(ID, Content) <-
60   .print("Received results for ", ID, ": ", Content).
```

The agent of Listing 21 is the most complex agent since it handles patient complaints, chooses which collection method to use, manages the collection, sends the lab order to the laboratory, and processes the final results.

While implementing these agents, we took note of the errors that we ran into. Table 3 summarizes these errors and their frequency in our initial implementation attempts.

These errors are generally typos or incomplete design changes that were not immediately obvious due to the distributed nature of the software; the producers and consumers of the information were separate programs, specified in separate files.

Except for the one syntax mistake, we consider that most of these bugs should be reduced or eliminated by the use of specification-driven tooling, such as Argus. The specification (BSPL in our case) keeps all of the names and schemas in a single place that can be checked for consistency, and then used to generate skeletons for implementation. Unfortunately, simple scaffolding only helps the initial implementation; after a design change, the developers must either modify the skeletons in place or regenerate them and migrate the entire implementation. Future tooling could minimize the cost of migration, but such tooling is out of scope for this project.

Protocols are conceptual objects, and agents must reason about them regardless of how they are implemented. Argus captures this reasoning via its code generation and adapter. Without Argus, Jason programmers must reason still about them, but without the kind of support that Argus provides.

8. Implementation

We have implemented the Argus communication model and AgentSpeak extensions using the Python-AgentSpeak implementation [45] of Jason. Our implementation is a library providing an adapter class that encapsulates both the protocol adapter and the Jason agent behavior, loading a protocol specified in BSPL and agent behavior specified in AgentSpeak.

Fig. 8 shows the Argus architecture. This architecture integrates and extends two existing architectures for building agents: BDI and protocol adapters. BDI agents have plans specified by their programming, a reasoner component that executes the plans, and a belief base that stores and enables queries for structured knowledge. Protocol adapters add interaction protocol support to an agent, using a protocol specification, a local store that records observed messages, a checker component that validates incoming/outgoing messages against the protocol specification, and an emitter and receiver that provide the network interface for message transmission.

Most Argus components are inherited from these prior architectures and their implementations are unchanged. These components are drawn as gray boxes. The extended components are highlighted with a red border: the Interpreter and Belief Base. The extended Interpreter component encompasses both the existing AgentSpeak reasoner and the protocol adapter's message checking. Because asynchronous messages can be received anytime, the integration enables the protocol checker to trigger AgentSpeak goals for handling message observations.

The Belief Base is extended by merging the Local State information about which messages have been observed. Because Argus inherits from existing implementations, the observed messages are stored twice: first as message objects in the protocol adapter's local state, where they are used by the protocol checker, and then copied as a belief in the AgentSpeak belief base. Thus, Argus extends the belief base with information about message observations.

Concretely, an agent is implemented as a Python module containing an instance of the adapter object, configured with the role it is to play and the IP addresses of the other agents.

Listing 22: Agent setup for Argus.

```
1 from argus import Adapter
2 from configuration import config, grading, Professor
```

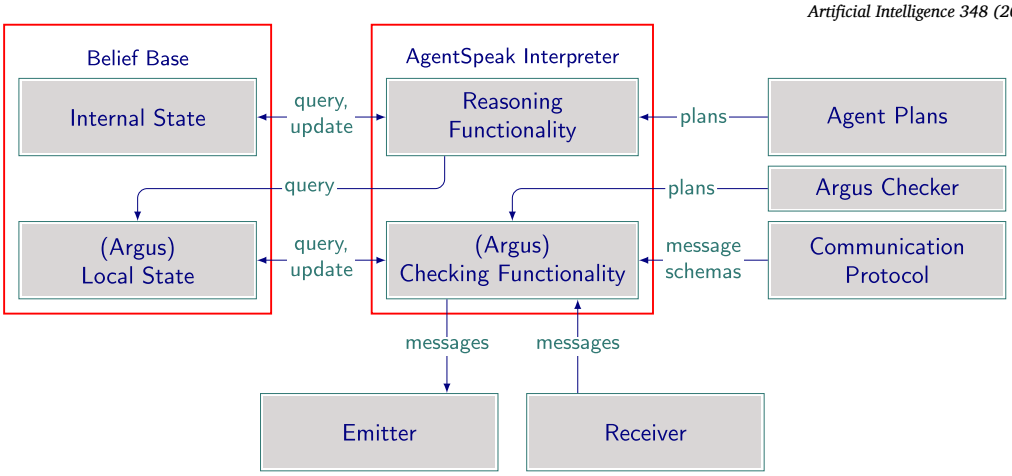


Fig. 8. The Argus architecture implemented in a BDI agent. The Argus adapter validates incoming and outgoing messages against the protocol and updates the local state. Plans expressed as traditional BDI artifacts are interpreted by the BDI engine. Argus blends into the BDI architecture because the Argus local state is reflected as a set of beliefs, as are already available to the BDI reasoner in the BDI architecture.

```
3 adapter = Adapter(Professor, grading, config, name="Pnin")
4 adapter.load_asl("pnin.asl")
5
6 if __name__ == "__main__":
7     print("Starting␣Pnin...")
8     adapter.start()
```

The adapter has an asynchronous update loop that interleaves the processing of messages with the steps of the Jason reasoner. By default, the Python-AgentSpeak reasoner runs until it has exhausted all its intentions and then halts. We added a signal which resumes the reasoner when a new message is received.

9. Discussion

Although some existing research combines BDI-style agent reasoning with normative abstractions such as commitments [46,47], agent programming languages have generally not kept up with advances in modeling communications. Argus addresses this gap by proposing a programming model that combines protocols with Jason's BDI-style agent programming abstractions. The specific form of protocols that we adopt here—information protocols—are fully declarative and match the representations on which rules, such as in Jason, may be stated. Through our choice of Jason as an exemplar of BDI-style programming abstractions, this paper indicates how protocols may be combined with other BDI architectures [48].

Improvements to Argus Currently, an agent can play at most one role.

This is a limitation not of BSPL, but of Argus. More than one agent cannot play a role; however, there can be many multiagent systems, which can simulate the effect of more than one agent per role, as demonstrated in the *Grading* scenario, where each student is in a separate MAS. Our approach works with BSPL, which is limited to one agent per role. Splee [49] addresses this limitation of BSPL with the concept of *set roles*, where a set of agents may play a role.

Argus guides programmers in implementing protocol-compliant agents. Using the adapter is an indispensable part of the programming model. Due to implementation errors, however, it remains possible for an agent to instruct the adapter to attempt an action that violates the protocol. Currently, these attempts are detected by the adapter, which silently abandons them. Although this ensures compliance, a better approach would be to log and signal the failure to the agent's internal reasoning.

Moreover, even if an agent is itself compliant, other agents may not be. Therefore, an agent could receive noncompliant messages. In such cases, Argus will again catch the error at runtime but silently discard the offending messages. The adapter could be extended to tag such messages as corrupted and notify the agent. The agent could, for example, use that information to update its estimate of the trustworthiness of the other agents.

In general, there is the problem of dealing with malicious agents in open systems. It is worth extending the adapter with support for signed messages and other measures for validation. However, even robust technical architectures such as those based on blockchain cannot guarantee correctness and perhaps shouldn't try to do so [50]. The problem is fundamentally sociotechnical; to address it effectively presupposes social components for effective governance [51].

Protocol-based programming models Several works address programming models based on role skeletons derived from protocols [52–54]. Typically, in these works, the protocol is specified via something like a UML sequence diagram, and role skeletons are expressed via rules or state machines that can be fleshed out with internal reasoning to realize an agent. However, in these approaches, there is

no formal model of the protocol and its messages. As such, these approaches don't support the kind of fine-grained reasoning about information we can use to (1) help the programmer via a programming model and (2) check each message for whether it (or its sender) respects the protocol semantics. Moreover, the protocols in these approaches are neither fully declarative nor supportive of asynchronous, order-insensitive communication channels.

Social meaning Communication meaning guides action by agents. For purposes of open MAS, social meaning approaches based on norms represent a compelling alternative to approaches such as KQML. Architecturally, the norms would be layered over protocols. Whereas a protocol specifies the basic communicative acts and inviolable operational constraints on their occurrence, norms capture how violable expectations between agents progress with the performance of those acts [17]. Recent efforts [55,56] have begun to address programming with commitments in a BDI framework.

JaCaMo JaCaMo [57] is a powerful framework for programming multiagent systems that brings together Jason and CArtAgO. (Recall the discussion of CArtAgO in Section 6.7.) JaCaMo models the environment via CArtAgO-based shared artifacts that agents may invoke operations on and use to coordinate their interactions, in a style reminiscent of Web services. JaCaMo currently does not support protocol-based communication, although Boissier et al. [57, footnote on p. 748] acknowledge its importance. Argus addresses this gap by improving the communicative foundation of Jason. Baldoni et al. [58] show how to extend and apply JaCaMo for reasoning about commitments. Argus could help place such approaches in decentralized multiagent systems, especially in light of new results demonstrating how to enact commitments over protocols [18,56].

The environment Modeling the environment of a multiagent system (e.g., shared resources) is crucial to systematically coordinating agents in a multiagent system [59,60]. In Argus, in accordance with decentralization, the requisite attributes of the environment would be captured in an information protocol [15]. In fact, the motivation behind information protocols is to capture the application domain, including the relevant resources and decision making by users. Returning to our example, parameters such as Question, Answer, and so on, may be thought of as shared resources in the domain. Some applications of multiagent systems may refer to physical incidents and resources in the environment, which may themselves be stateful. For example, in an emergency response application, we might see a fire (with states raging, dying, out, and so on) and a fire alarm (off, armed, triggered, and so on). However, the states of such physical entities would have to be modeled in the information exchanged via a protocol. For example, upon noticing a fire in her building and that the fire alarm had not gone off, a resident could send a message to emergency services that a fire has erupted without triggering the fire alarm. Naturally, it is the communicated information that counts toward the state of the multiagent system. An interesting problem is how to resolve discrepancies between the physical state and the state of the multiagent system [61].

An agent can act upon information from diverse sources in its environment. What information should we capture in a protocol and what should be left to agents' internal states? The following informal rule answers the question and helps the engineering of modular, loosely coupled multiagent systems.

If a source of some relevant information is interactive, then the information should be modeled via parameters in the protocol. If the source is not interactive, it just affects some agent's internal state. The agent may decide to put it into a protocol enactment by binding it to parameters in the protocol.

Session types Session types [62] specify a protocol from a global perspective in terms of the emissions and receptions on channels between agents, with choice operators to indicate which agent drives the interaction at any point in time. As such, the meaning of a message in a session type derives from its position in the sequence of transmissions, in contrast with BSPL and information protocols, where the meaning of a message (its parameters and their relationships) determines when the emission of that message is enabled. Thus, session types are susceptible to several of the same criticisms we raised against previous BDI communication models in Section 6: specifically, the lack of semantic correlation and selective reception. Chopra et al. [15] presents a substantial critique of session types.

Although session types may be a reasonable model for existing BDI agent implementations, Argus shows how information-based communication models can be a good fit for the logic programming style of BDI agents, better support agent autonomy, and reduce coupling.

Data-driven approaches Information protocols provide abstractions that fit well with other data-driven approaches for interaction and business process modeling [63,64]. These approaches typically enhance workflows with operations on databases and support high-level abstractions for specifying an agent's internal reasoning. Argus, by providing a connection with Jason, can further help relate data-driven and rule-based BDI approaches. Conceivably, the more flexible parts of agents—those dealing with interaction—could be generated in Jason (as we do in this paper), whereas the rules corresponding to the internal reasoning could be generated from a data-driven specification.

Microservices and the Internet of Things (IoT) Both are popular industry paradigms that emphasize decentralization. In the microservices paradigm [65], the ideal is that the system is constituted from autonomous microservices that interact via asynchronous messaging. Such a system conception is essentially multiagent. Today, however, the design of microservices is not based on protocols. BSPL and Argus demonstrate how a microservices-based system could be modeled in terms of protocols and implemented accordingly. Khadse et al. [66] represents a start in this direction. The IoT often features decentralized applications that feature extreme asynchrony: a sensor could come alive, fire off a transmission, and go to sleep. We have initiated exploration of this direction via a Node-RED implementation of BSPL [16].

Rule-based programming and testing Protocols and protocol-based programming, as Argus supports, lead to greater decoupling between agents and between an agent and a protocol. In addition, protocols help modelers and programmers avoid ad hoc functions, such as S_M, that operate outside of the BDI structure. Further, the generated code facilitates an agent programmer's primary task by providing clear points for plugging in an agent's internal reasoning. The concomitant increased clarity in code structure and reduced programmer effort can help address the difficult task of testing BDI programs [67] by focusing testing on the internal reasoning of agents. In rule-based programming, errors involving pattern matching are potentially easy to make and hard to find. For example, any difference in parameter ordering would bind parameters to the wrong values, and missing or extra parameters would silently prevent the plan from matching at all. Code generation even of agent skeletons, as we demonstrate here, helps alleviate these problems.

Going further, ideally, the programming model should make it clear—via some form of typing—the parameters of an enabled message that an agent's plan needs to bind. Another future direction we are considering is generalizing to an enablement-based programming model to handle sets of possible messages. A benefit would be that an agent could optimize its interactions if it could evaluate a set of alternatives in one shot.

Uncertainty A general problem though is how to model uncertainty in the information (and beliefs) being generated by enacting protocols and what that means for protocol enactment. For example, the resident may claim a fire with 80% certainty. Would that *enable* the emergency services to dispatch a firefighting team? Uncertainty is widely considered important in cognitive reasoning. Yao et al. [68] study the problem of intention selection under the uncertainty of beliefs. Techniques from this body of work could possibly be adapted for protocols.

Multiagent organizations A strength of several agent-oriented approaches is their focus on modeling multiagent systems via organizational notions. Ferber and Gutknecht [69] structure an organization into roles and groups based on the idea that only agents within a group may communicate. Hübner et al. [70] define organizational goals and agents commit to missions that promote those goals. Moreover, obligations and permissions are attached to the roles that agents adopt. Cossentino et al. [71] define organizations of agents via tasks, individual goals, and collective goals. Whereas this paper does not deal with organizational notions, they are crucial for enabling programming at a higher level of abstraction.

10. Conclusions and future directions

This paper has demonstrated that communication protocols based on information flow can be combined with a BDI approach in a way that highlights the benefits of protocols without compromising the benefits of BDI programming. The paper has also shown the net benefits of decoupling agents in multiagent systems and, through the avoidance of hidden functions, the development of purer BDI agents than is possible traditionally.

Fig. 4 shows the Argus operations of EMIT and RECEIVE placed with the Jason reasoning cycle. In essence, with the exception of Jason's transition from ExecInt to ProcMsg, which is removed in Argus, all other transitions are kept. The ExecInt to ClrInt and ProcMsg to SelEv transitions are changed to accommodate the BSPL semantics, as explained below. We did not find theorems about the soundness and completeness of the operational semantics of Jason. Therefore, we content ourselves with the hint of an informal argument here. Assuming Jason's semantics is sound and complete, we would be able to decide for each agent whether it would eventually return to its initial (and final) "state," i.e., the ProcMsg step. In addition, let's suppose the BSPL protocol being implemented is *safe* and *live* (these properties, respectively, being the interactional analogs of soundness and completeness) [14,8]. Then, the agents in a multiagent system based on Argus would repeatedly return to their respective ProcMsg steps. We defer the formulation of the requisite notions of soundness and completeness and their establishment to future research.

The ideas advanced in this paper set the stage for further research. An important direction concerns the development of new programming models that take better advantage of protocols and BDI representations to improve the modularity and maintainability of implementations of agents in multiagent systems. A related challenge concerns support for application-level fault tolerance: initial work uses interaction protocols to derive agent *expectations* that indicate failures when unmet [72], but uses an ad hoc policy language for specifying recovery policies; a BDI-based approach, building on Orpheus [55] or Azorus [56], might offer improved flexibility and simplify establishing consistency. Extending the Argus programming model to support exploiting group-oriented communication abstractions provided by platforms such as Janus [73] is also an interesting direction.

The architectural style underlying BSPL, called Local State Transfer (LoST), has been compared to the REST architectural style for Web services [27]. LoST is indeed RESTful in spirit but goes beyond REST in that LoST (due to BSPL) promotes decentralized architectures, whereas REST is client-server. HATEOAS, a REST constraint, promotes a view of Web services as state machines, where resource representations returned by a service determine the operations that can be invoked on it next. Conceptually, an Argus agent (following LoST) does something similar: Its local state, as maintained by the adapter, determines which among the set of enabled actions it can perform next.

Kiko [25] and Azorus [56] make the notion of enabled actions explicit in their programming models. Since most applications today run on the Web, it would be interesting to explore how decentralized applications based on interaction protocols could be realized on top of the Web. Fluid [74] takes a step in that direction.

Our model of how much the parties trust each other can influence MAS design, for example, by introducing mediators, audits, and so on [75,76]. Work on identifying and applying trust-promoting organizational patterns to protocols would be valuable. However, it is worth keeping in mind that no MAS design can obviate trust. Agents may always engage in untrustworthy behaviors by violating protocols. Even protocol-compliant agents may violate the relevant norms. Argus helps engineer protocol-compliant agents, but it

makes no assumption that the agents are trustworthy. Work on estimating trust based on actual enactments with the aim of more judicious decision making would also be valuable.

The purpose behind Lamport's influential idea of *potential causality* (as captured in the *happens before* relation) is to infer an ordering of events in a distributed system [77]. By contrast, one of our guiding principles (as exemplified by BSPL) is that any event ordering or occurrence requirement falls out from the specification of causality in protocols and physical causality (sends before receives).

Our approach captures true causality between events and enables a causal analysis of protocols [8,78,79], which helps make verification tractable in many cases. New types of causal analysis with additional benefits would be a direction with rich dividends.

We need automated tools for porting legacy Jason implementations to Argus. These tools could assist humans in abstracting out protocols, assuming agent implementations are available. These tools could also help us repurpose Jason's plans for Argus by replacing each use of a KQML performative, including Ask and its variants, with corresponding messages or protocols.

In principle, an agent's local state is ever-growing, which obviously has costs and efficiency implications. As in real life, the way to address such concerns is by introducing methods for archiving *old* state (as captured by some heuristic) to low-cost storage. Based on the assumption that old state would be rarely needed, this would save costs and make querying the agent's "working" state more efficient. Identifying patterns of archival policies and supporting them in an agent programming model would be another interesting direction.

11. Reproducibility

The entire Argus codebase and all examples are available online at https://gitlab.com/masr/bspl, where the Argus AgentSpeak components have been merged into the protocol adapter. The grading scenario is available there in the `scenarios/grading` directory.

CRediT authorship contribution statement

Samuel H. Christie: Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization. **Munindar P. Singh:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Conceptualization. **Amit K. Chopra:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Thanks to the NSF (grant IIS-1908374) and EPSRC (grant EP/N027965/1) for partial support.

Appendix A

Listing 23: An alternative grading protocol that allows PROFESSOR to declare the *end* of a test (meaning that there will be no more *challenges*) and STUDENT to *resign* from a test (meaning that there will be no more *responses*).

```
More-Regimented-Grading {
  roles Professor, Student, TA
  parameters out TID key, out Report
  private QID, Question, Solution, Answer, Grade, NumChallenges, NumResponses, Done, Finished

  Professor ↦ Student: beginTest[out TID key]
  Professor ↦ Student: challenge[in TID key, out QID key, out Question, nil Done, nil Finished]
  Professor ↦ Student: end[in TID key, out NumChallenges, out Done]

  Professor ↦ TA: rubric[in TID key, in QID key, out Solution]
  Student ↦ TA: response[in TID key, in QID key, in Question, out Answer, nil Finished]
  Student ↦ Professor: resign[in TID key, out NumResponses, out Finished]
  TA ↦ Professor: result[in TID key, in QID key, in Answer, in Solution, out Grade]
  Professor ↦ Student: performance[in TID key, out Report]
}
```

Listing 23 gives an alternative grading protocol that enables students to resign from a test and professors to declare the end of a test. PROFESSOR cannot send any more *challenges* after sending *end* or receiving *resign* because they bind parameters Done and Finished, respectively, that must be unbound (⌜nil⌝) in *challenge*. STUDENT is sender of both *response* and *resign*.

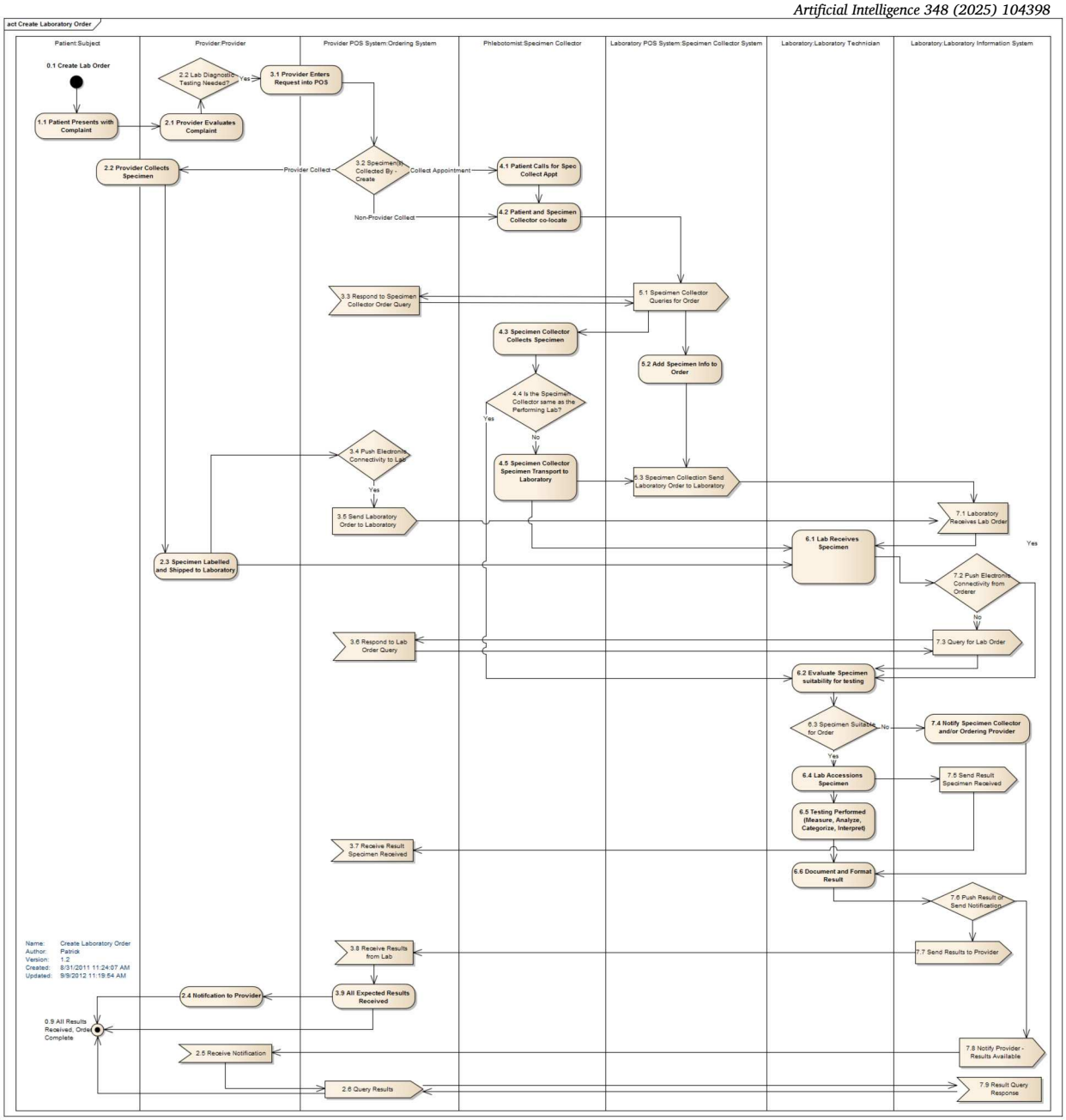


Fig. 9. The original Create Lab Order workflow [43]. The workflow is complex and informally specified, which makes implementation difficult. This workflow is available at http://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification.

STUDENT cannot send any more *responses* after sending *resign* because *resign* binds Finished, which must be unbound to send *response*. However, STUDENT may receive and *respond* to a *challenge* even after receiving *end*, illustrating how BSPL elegantly deals with message reordering in the network.

Notice that the parameter line is different compared to Listing 4: It refers to only the test-level, that is, TID-level parameters. This is necessary to ensure the liveness of the protocol because a test may terminate with no questions asked (PROFESSOR sends *end* without sending any *challenges*) or not all solutions offered (STUDENT sends *resign* without sending a *response* to every *challenge*). In the former case, there will be no QID binding; in the latter case, there will be no Grade binding. PROFESSOR can send *performance* anytime after the test begins, generating a binding for Report, which means there is always a path to completion for protocol enactments, although the normative expectation will be that the *Report* accounts for all of the STUDENT's timely *responses*. BSPL is silent on the normative expectations, as on trust and any disputes that may arise (e.g., about the timeliness of a *response*); however, it gives an operational substrate for layering on these considerations.

Data availability

We have included a link to a public code repository in the Reproducibility section of the paper.

References

[1] A.K. Chopra, M.P. Singh, From social machines to social protocols: software engineering foundations for sociotechnical systems, in: Proceedings of the 25th International World Wide Web Conference, ACM, Montréal, 2016, pp. 903–914.
[2] M.N. Huhns, N. Jacobs, T. Ksiezyk, W.-M. Shen, M.P. Singh, P.E. Cannata, Integrating enterprise information models in Carnot, in: Proceedings of the International Conference on Intelligent and Cooperative Information Systems (ICICIS), IEEE, Rotterdam, 1993, pp. 32–42.
[3] M.P. Singh, Agent communication languages: rethinking the principles, IEEE Comput. 31 (1998) 40–47, https://doi.org/10.1109/2.735849.
[4] M. Baldoni, C. Baroglio, A.K. Chopra, N. Desai, V. Patti, M.P. Singh, Choice, interoperability, and conformance in interaction protocols and service choreographies, in: Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Budapest, 2009, pp. 843–850.
[5] M. Winikoff, W. Liu, J. Harland, Enhancing commitment machines, in: Proceedings of the 2nd International Workshop on Declarative Agent Languages and Technologies (DALT), in: Lecture Notes in Artificial Intelligence, vol. 3476, Springer, Berlin, 2005, pp. 198–220.
[6] A.K. Chopra, M.P. Singh, Contextualizing commitment protocols, in: Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems, ACM Press, Hakodate, Japan, 2006, pp. 1345–1352.
[7] A.U. Mallya, M.P. Singh, An algebra for commitment protocols, J. Auton. Agents Multi-Agent Syst. 14 (2007) 143–163, https://doi.org/10.1007/s10458-006-7232-1.
[8] M.P. Singh, S.H. Christie V, Tango: declarative semantics for multiagent communication protocols, in: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Online, 2021, pp. 391–397.
[9] P. Yolum, Design time analysis of multiagent protocols, Data Knowl. Eng. 63 (2007) 137–154, https://doi.org/10.1016/j.datak.2006.12.001.
[10] R.H. Bordini, J.F. Hübner, Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions), in: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI), in: Frontiers in Artificial Intelligence and Applications, vol. 215, IOS Press, Lisbon, 2010, pp. 635–640.
[11] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos, Tropos: an agent-oriented software development methodology, J. Auton. Agents Multi-Agent Syst. 8 (2004) 203–236, https://doi.org/10.1023/B:AGNT.0000018806.20944.ef.
[12] J. Harland, D.N. Morley, J. Thangarajah, N. Yorke-Smith, Aborting, suspending, and resuming goals and plans in BDI agents, J. Auton. Agents Multi-Agent Syst. 31 (2017) 288–331, https://doi.org/10.1007/s10458-015-9322-4.
[13] M.P. Singh, 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), IFAAMAS, Taipei, 2011, pp. 491–498.
[14] M.P. Singh, Semantics and verification of information-based protocols, in: Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), IFAAMAS, Valencia, Spain, 2012, pp. 1149–1156.
[15] A.K. Chopra, S.H. Christie V, M.P. Singh, An evaluation of communication protocol languages for engineering multiagent systems, J. Artif. Intell. Res. 69 (2020) 1351–1393, https://doi.org/10.1613/jair.1.12212.
[16] S.H. Christie V, D. Smirnova, A.K. Chopra, M.P. Singh, Protocols over things: a decentralized programming model for the Internet of Things, IEEE Comput. 53 (2020) 60–68, https://doi.org/10.1109/MC.2020.3023887.
[17] T.C. King, A. Günay, A.K. Chopra, M.P. Singh, Tosca: operationalizing commitments over information protocols, in: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Melbourne, 2017, pp. 256–264.
[18] M.P. Singh, A.K. Chopra, Clouseau: generating communication protocols from commitments, in: Proceedings of the 34th Conference on Artificial Intelligence (AAAI), AAAI Press, New York, 2020, pp. 7244–7252.
[19] M. Dastani, 2APL: a practical agent programming language, J. Auton. Agents Multi-Agent Syst. 16 (2008) 214–248, https://doi.org/10.1007/s10458-008-9036-y.
[20] M. Dastani, M.B. van Riemsdijk, J.-J.C. Meyer, Programming multi-agent systems in 3APL, in: R.H. Bordini, M. Dastani, J. Dix, A. El, Fallah Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, in: Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, Springer, Boston, 2005, pp. 39–67.
[21] K.V. Hindriks, J. Dix, GOAL: a multi-agent programming language applied to an exploration game, in: O. Shehory, A. Sturm (Eds.), Agent-Oriented Software Engineering – Reflections on Architectures, Methodologies, Languages, and Frameworks, Springer, Berlin, 2014, pp. 235–258.
[22] H. Chalupsky, T. Finin, R. Fritzson, D. McKay, S. Shapiro, G. Wiederhold, An Overview of KQML: A Knowledge Query and Manipulation Language, TR, University of Maryland Computer Science Department, Baltimore, 1992.
[23] J.L. Austin, How to Do Things with Words, Clarendon Press, Oxford, 1962.
[24] R. Vieira, Á.F. Moreira, M.J. Wooldridge, R.H. Bordini, On the formal semantics of speech-act based communication in an agent-oriented programming language, J. Artif. Intell. Res. 29 (2007) 221–267, https://doi.org/10.1613/jair.2221.
[25] S.H. Christie V, M.P. Singh, A.K. Chopra, Kiko: programming agents to enact interaction protocols, in: Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, London, 2023, pp. 1154–1163.
[26] A. Günay, A.K. Chopra, Stellar: a programming model for developing protocol-compliant agents, in: Proceedings of the 6th International Workshop on Engineering Multi-Agent Systems (EMAS), in: Lecture Notes in Computer Science, vol. 11375, Springer, Stockholm, 2018, pp. 117–136.
[27] M.P. Singh, LoST: Local State Transfer—an architectural style for the distributed enactment of business protocols, in: Proceedings of the 9th IEEE International Conference on Web Services (ICWS), IEEE Computer Society, Washington, DC, 2011, pp. 57–64.
[28] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60–61 (2004) 17–139, https://doi.org/10.1016/j.jlap.2004.05.001.
[29] N. Carriero, D. Gelernter, Linda in context, Commun. ACM 32 (1989) 444–458, https://doi.org/10.1145/63334.63337.
[30] A.K. Chopra, A. Artikis, J. Bentahar, M. Colombetti, F. Dignum, N. Fornara, A.J.I. Jones, M.P. Singh, P. Yolum, Research directions in agent communication, ACM Trans. Intell. Syst. Technol. 42 (2013) 20:1–20:23, https://doi.org/10.1145/2438653.2438655.
[31] D. Vanderveken, Meaning and Speech Acts, Principles of Language Use, vol. 1, Cambridge University Press, Cambridge, United Kingdom, 1990.
[32] M.P. Singh, A social semantics for agent communication languages, in: Proceedings of the 1999 IJCAI Workshop on Agent Communication Languages, in: Lecture Notes in Artificial Intelligence, vol. 1916, Springer, Berlin, 2000, pp. 31–45.
[33] A. Artikis, M.J. Sergot, J.V. Pitt, Specifying norm-governed computational societies, ACM Trans. Comput. Log. 10 (2009) 1:1–1:42, https://doi.org/10.1145/1459010.1459011.
[34] A.K. Chopra, M.P. Singh, Custard: computing norm states over information stores, in: Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Singapore, 2016, pp. 1096–1105.
[35] M.P. Singh, Semantical considerations on dialectical and practical commitments, in: Proceedings of the 23rd Conference on Artificial Intelligence (AAAI), AAAI Press, Chicago, 2008, pp. 176–181, https://cdn.aaai.org/AAAI/2008/AAAI08-028.pdf/.
[36] A.K. Chopra, M.P. Singh, Specifying and applying commitment-based business patterns, in: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Taipei, 2011, pp. 475–482.
[37] R. Fagin, J.Y. Halpern, Y. Moses, M.Y. Vardi, Reasoning About Knowledge, MIT Press, Cambridge, Massachusetts, 1995.

[38] D.R. Reddy, L.D. Erman, R.D. Fenneli, R.B. Neely, The Hearsay speech understanding system: an example of the recognition process, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, William Kaufmann, Stanford, California, 1973, pp. 185–193, http://ijcai.org/Proceedings/73/Papers/021.pdf.

[39] L.D. Erman, F. Hayes-Roth, V.R. Lesser, R. Reddy, The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty, ACM Comput. Surv. 12 (1980) 213–253, https://doi.org/10.1145/356810.356816.

[40] N. Carriero, D. Gelernter, Coordination languages and their significance, Commun. ACM 35 (1992) 97–107, https://doi.org/10.1145/129630.376083.

[41] A. Ricci, M. Piunti, M. Viroli, A. Omicini, Environment programming in CArtAgO, in: R.H. Bordini, M. Dastani, J. Dix, A.E.F. Seghrouchni (Eds.), Multi-Agent Programming, Languages, Tools and Applications, Springer, Dordrecht, Netherlands, 2009, pp. 259–288.

[42] M. van Steen, A.S. Tanenbaum, A brief introduction to distributed systems, Computing 98 (2016) 967–1009, https://doi.org/10.1007/s00607-016-0508-7.

[43] HL7, Laboratory Order Conceptual Specification, Technical Report, Health Level Seven International, Ann Arbor, Michigan, 2013, http://wiki.hl7.org/index.php?title=Laboratory_Order_Conceptual_Specification. (Accessed 24 November 2023).

[44] S.H. Christie V, A.K. Chopra, M.P. Singh, Compositional correctness in multiagent interactions, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Stockholm, 2018, pp. 1159–1167.

[45] N. Fiekas, Jason-style AgentSpeak for Python, https://github.com/niklasf/python-agentspeak, 2021, https://github.com/niklasf/python-agentspeak. (Accessed 31 August 2021).

[46] A. Günay, M. Winikoff, P. Yolum, Dynamically generated commitment protocols in open systems, J. Auton. Agents Multi-Agent Syst. 29 (2015) 192–229, https://doi.org/10.1007/s10458-014-9251-7.

[47] F. Meneguzzi, M.C. Magnaguagno, M.P. Singh, P.R. Telang, N. Yorke-Smith, Goco: planning expressive commitment protocols, J. Auton. Agents Multi-Agent Syst. 32 (2018) 459–502, https://doi.org/10.1007/s10458-018-9385-0.

[48] L. de Silva, F. Meneguzzi, B. Logan, BDI agent architectures: a survey, in: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Online, 2020, pp. 4914–4921.

[49] A.K. Chopra, S.H. Christie V, M.P. Singh, Splee: a declarative information-based language for multiagent interaction protocols, in: Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, São Paulo, 2017, pp. 1054–1063.

[50] M.P. Singh, A.K. Chopra, Computational governance and violable contracts for blockchain applications, IEEE Comput. 53 (2020) 53–62, https://doi.org/10.1109/MC.2019.2947372.

[51] M.P. Singh, Cybersecurity as an application domain for multiagent systems, in: Proceedings of the 14th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Istanbul, Blue Sky Ideas Track, 2015, pp. 1207–1212.

[52] C. Rooney, R.W. Collier, G.M.P. O'Hare, VIPER: a VIsual protocol EditoR, in: Proceedings of the 6th International Conference on Coordination Models and Languages COORDINATION, in: Lecture Notes in Computer Science, vol. 2949, Springer, Pisa, 2004, pp. 279–293.

[53] N. Desai, A.U. Mallya, A.K. Chopra, M.P. Singh, Interaction protocols as design abstractions for business processes, IEEE Trans. Softw. Eng. 31 (2005) 1015–1027, https://doi.org/10.1109/TSE.2005.140.

[54] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Agent-oriented model-driven development for JADE with the JADEL programming language, Comput. Lang. Syst. Struct. 50 (2017) 142–158, https://doi.org/10.1016/j.cl.2017.06.001.

[55] M. Baldoni, S.H. Christie V, M.P. Singh, A.K. Chopra, Orpheus: engineering multiagent systems via communicating agents, in: Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI), AAAI, Philadelphia, 2025, pp. 23135–23143.

[56] A.K. Chopra, M. Baldoni, S.H. Christie V, M.P. Singh, Azorus: commitments over protocols for BDI agents, in: Proceedings of the 24th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Detroit, 2025, pp. 490–499.

[57] O. Boissier, R.H. Bordini, J.F. Hübner, A. Ricci, Dimensions in programming multi-agent systems, Knowl. Eng. Rev. 34 (2019) e2, https://doi.org/10.1017/S026988891800005X.

[58] M. Baldoni, C. Baroglio, F. Capuzzimati, R. Micalizio, Commitment-based agent interaction in JaCaMo+, Fundam. Inform. 159 (2018) 1–33, https://doi.org/10.3233/FI-2018-1656.

[59] D. Weyns, A. Omicini, J. Odell, Environment as a first class abstraction in multiagent systems, J. Auton. Agents Multi-Agent Syst. 14 (2007) 5–30, https://doi.org/10.1007/s10458-006-0012-0.

[60] A. Ricci, A. Ciortea, S. Mayer, O. Boissier, R.H. Bordini, J.F. Hübner, Engineering scalable distributed environments and organizations for MAS, in: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS, Montréal, 2019, pp. 790–798.

[61] T. Leesatapornwongsa, A. Sengupta, M.S. Ardekani, G. Petri, C.A. Stuardo, Transactuations: where transactions meet the physical world, ACM Trans. Comput. Syst. 36 (2020) 13:1–13:31, https://doi.org/10.1145/3380907.

[62] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (2016) 9:1–9:67, https://doi.org/10.1145/2827695.

[63] R. De Masellis, C.D. Francescomarino, C. Ghidini, M. Montali, S. Tessaris, Add data into business process verification: bridging the gap between theory and practice, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, 2017, pp. 1091–1099.

[64] M. Montali, D. Calvanese, G.D. Giacomo, Verification of data-aware commitment-based multiagent system, in: Proceedings of the 13th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), IFAAMAS/ACM, Paris, 2014, pp. 157–164.

[65] Microsoft, Microservice architecture style, https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices, 2019. (Accessed 16 June 2021).

[66] A.K. Khadse, S.H. Christie V, A.K. Chopra, M.P. Singh, Protocol-based engineering of microservices, in: Proceedings of the 11th International Workshop on Engineering Multi-Agent Systems (EMAS), London, in: Lecture Notes in Artificial Intelligence, vol. 14378, 2023, pp. 61–77.

[67] M. Winikoff, S. Cranefield, On the testability of BDI agent systems, J. Artif. Intell. Res. 51 (2014) 71–131, https://doi.org/10.1613/jair.4458.

[68] Y. Yao, N. Alechina, B. Logan, J. Thangarajah, Intention progression using quantitative summary information, in: Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), ACM, London, 2021, pp. 1416–1424.

[69] J. Ferber, O. Gutknecht, A meta-model for the analysis and design of organizations in multi-agent systems, in: Proceedings of the Third International Conference on Multiagent Systems, IEEE, 1998, pp. 128–135.

[70] J.F. Hübner, J.S. Sichman, O. Boissier, Developing organised multiagent systems using the MOISE$^+$ model: programming issues at the system and agent levels, Int. J. Agent-Oriented Softw. Eng. 1 (2007) 370–395, https://doi.org/10.1504/IJAOSE.2007.016266.

[71] M. Cossentino, N. Gaud, V. Hilaire, S. Galland, A. Koukam, ASPECS: an agent-oriented software process for engineering complex systems, J. Auton. Agents Multi-Agent Syst. 20 (2010) 260–304, https://doi.org/10.1007/s10458-009-9099-4.

[72] S.H. Christie V, A.K. Chopra, M.P. Singh, Mandrake: multiagent systems as a basis for programming fault-tolerant decentralized applications, J. Auton. Agents Multi-Agent Syst. 36 (2022) 16:1–16:30, https://doi.org/10.1007/s10458-021-09540-8.

[73] S. Galland, S. Rodriguez, N. Gaud, Run-time environment for the SARL agent-programming language: the example of the Janus platform, Future Gener. Comput. Syst. 107 (2020) 1105–1115, https://doi.org/10.1016/j.future.2017.10.020.

[74] A.K. Chopra, M.P. Singh, Fluid: social norms–based multiagent systems on the web, in: Pre-Proceedings of the 13th International Workshop on Engineering Multi-Agent Systems (EMAS), Detroit, 2025, pp. 1–18.

[75] E. Paja, A.K. Chopra, P. Giorgini, Trust-based specification of sociotechnical systems, Data Knowl. Eng. 87 (2013) 339–353, https://doi.org/10.1016/j.datak.2012.12.005.

[76] A.K. Chopra, N. Oren, S. Modgil, N. Desai, S. Miles, M. Luck, M.P. Singh, Analyzing contract robustness through a model of commitments, in: Proceedings of the 11th International Workshop on Agent Oriented Software Engineering (AOSE 2010), in: Lecture Notes in Computer Science, vol. number 6788, Springer, Toronto, 2011, pp. 17–36.
[77] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565, https://doi.org/10.1145/359545.359563.
[78] M.P. Singh, S.H. Christie V, A.K. Chopra, Langshaw: declarative interaction protocols based on sayso and conflict, in: Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Jeju, Korea, 2024, pp. 202–210.
[79] A.K. Chopra, S.H. Christie V, M.P. Singh, Requirement patterns for multiagent interaction protocols, in: Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI), IJCAI, Montréal, 2025.