Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange

Davide Ancona
DISI, Università di Genova, Italy
davide@disi.unige.it

Viviana Mascardi
DISI, Università di Genova, Italy
mascardi@disi.unige.it

Jomi F. Hübner
DSC, Univ. de Blumenau, Brazil
jomi@inf.furb.br

Rafael H. Bordini†
Univ. of Liverpool, L69 3BX, U.K.
R.Bordini@csc.liv.ac.uk

Abstract

This paper brings together two recent contributions to the area of declarative agent-oriented programming, made feasible in practice by the recent introduction of an interpreter for a BDI programming language. The work on Coo-BDI has proposed an approach to plan exchange which applies to BDI agents in general. The other contribution is the introduction of special illocutionary forces for plan exchange between AgentSpeak agents. This has been implemented in Jason, an interpreter for an extended version of that language. This paper shows how the elaborate plan exchange mechanism of Coo-BDI can be used by AgentSpeak agents implemented with Jason. It also discusses an application on PDA-based multi-media presentations in museum visits for which plan exchange is relevant.

1. Introduction

Various agent-oriented programming languages have appeared in the literature since Shoham's seminal work [12]. Typically, these languages concentrate on the programming of one individual autonomous agent, leaving it completely for the user to work out ways of developing multi-agent system where those agents interact. However, the advantage of the agent-oriented approach to software development resides precisely in the fact that computational systems for dynamic and complex scenarios can be designed more easily by relying on various autonomous agents coordinating their actions.

On the other hand, agent-oriented software engineering approaches (see, e.g., [14]) have focussed mainly on societal aspects of building multi-agent systems, such as groups and roles, rather than the development of individual agents and their cognitive aspects. One of the first agent programming languages to move towards a more societal approach is Concurrent METATEM, where agents are specified directly in linear temporal logic: see their recent work on a general notion of groups [7]. There are sophisticated ways of developing teams of agents, such as STEAM [10] for example, but these are not programming languages as such, and do not have the same formal basis (e.g., formal semantics) as do most agent-oriented programming languages.

In this work, we are interested in an important aspect of the necessary support for future work on programming teams of agents where each agent is programmed in an agent-oriented programming language. We consider here the means for autonomous agents to exchange plans, so that agents can improve their abilities by obtaining plans from other agents that might have the specific know-how in question. To the best of our knowledge, this issue has not been dealt with by other agent-oriented programming languages such as Dribble [13], 3APL [5], and ConGolog [6].

Our starting point is Coo-BDI, an approach to cooperation for BDI agents by plan exchange, developed by Ancona and Mascardi and reported in [1]. In this paper, that approach is applied specifically to an extension of a BDI agent-oriented programming language called AgentSpeak(L), in the context of ***Jason*** [3], an interpreter for that extended language recently made available. AgentSpeak(L) was originally devised by Rao [11] and later extended and formalised by Bordini and colleagues [2, 4]. In [9], the operational semantics of AgentSpeak(L) was extended to account for speech-act based communication, including special illocutionary forces for communicating plans. ***Jason*** implements the operational semantics given in [4] as well as the extensions in [9]. This gives the necessary formal and practical basis for plan exchange among BDI agent in the way required by the approach presented in this paper.

† Currently at the Department of Computer Science, University of Durham, DH1 3LE, U.K. E-mail: R.Bordini@durham.ac.uk.

This paper is organised as follows. In the next section, we summarise Coo-BDI, an approach to plan exchange among BDI agents. Section 3 describes ***Jason***, a fully-fledged interpreter for an extended version of AgentSpeak(L). In both sections, only the features that are relevant for the work here are discussed. Then, in Section 4 we describe a scenario which shows the importance of plan exchange for BDI agents; the scenario is that of a PDA used to assist visitors in museums, art galleries, etc. This scenario is used in Section 5 where we present Coo-AgentSpeak, the instantiation of Coo-BDI for AgentSpeak in particular, and discuss how these ideas can be elegantly integrated into ***Jason***. The paper assumes that the reader is familiar with the BDI architecture as well as the AgentSpeak language.

2. Coo-BDI

Coo-BDI (Cooperative BDI [1]) extends traditional BDI agent-oriented programming languages in many respects. As in the traditional BDI setting, Coo-BDI agents are characterised by an event queue, a mailbox, a plan library, a belief base, and a set of intentions. The main extensions of Coo-BDI involve the introduction of *cooperation* among agents for the retrieval of external plans for a given triggering event; the extension of *plans* with "access specifiers"; the extension of *intentions* to take into account the external plan retrieval mechanism; and the modification of the *Coo-BDI engine* (i.e., the interpreter) to cope with all these issues.

The version of Coo-BDI described here is different from the one described in [1] for two main reasons:

Enhancement of the flexibility and expressive power of Coo-BDI. The granularity of the cooperative strategy has been refined, so that it becomes possible to apply different strategies to different kinds of plan, instead of having a unique cooperative policy for all the plans. Plans have been extended so as to include the plan source (namely, the agent that originally "owned" the plan).

Compliance with AgentSpeak and *Jason*. In order to facilitate the integration of Coo-BDI and AgentSpeak, implemented by means of the ***Jason*** interpreter, the distinction between "external events" and "desires" discussed in [1] has been abandoned: only AgentSpeak "events" are considered now. Plans have been adapted to the syntax supported by ***Jason***: constructs such as the invariant, and success and failure actions have been dropped, as well as branches in bodies. Messages exchanged between agents have also been modified in order to comply with the format supported by AgentSpeak in ***Jason***.

The description of some issues of Coo-BDI which are not particularly relevant for this work have been omitted or simplified for the sake of conciseness and clarity. We now discuss the Coo-BDI extensions in more detail.

Cooperation strategy. The cooperation strategy of an agent `Ag` includes the set of agents which are expected to cooperate with `Ag`, the plan retrieval policy, and the plan acquisition policy. The cooperation strategy may evolve during time, allowing greater flexibility and autonomy to the agents, and is modelled by three predicates:

- `trusted(Te,TrustedAgentSet)`, where `Te` is a (not necessarily ground) triggering event and `TrustedAgentSet` is the set of agents that `Ag` can contact in order to obtain plans that are relevant for `Te`.
- `retrievalPolicy(Te,Retrieval)`, where `Retrieval` may assume the values `always` and `noLocal`, meaning that relevant plans for the triggering event `Te` must always be retrieved from other agents, or only when no local relevant plans are available, respectively.
- `acquisitionPolicy(Te,Acquisition)`, where `Acquisition` may assume the values `discard`, `add`, and `replace` meaning that, when a relevant plan for `Te` is retrieved from a trusted agent, it must be discarded after its current use, or added to the plan library, or used to update the plan library by replacing all existing plans triggered by `Te`.

Plans. Besides the standard components which constitute BDI plans, Coo-BDI plans also have a *source* which refers to the agent from which the plan was obtained, and an *access specifier* which determines the set of agents with which the plan can be shared. The source may assume two values: `self` (the agent itself owns the plan) and `Ag` (the plan was originally from agent `Ag`). The access specifier may assume three values: `private` (the plan cannot be shared), `public` (the plan can be shared with any agent) and `only`*(TrustedAgentSet)* (the plan can be shared only with the agents in *TrustedAgentSet*).

Intentions. Intentions are characterised by the "standard" components plus those introduced due to the external-plan retrieval mechanism:

- the relevant plans which have already been collected for managing of the current goal; and
- the set of identifiers of those agents which are still expected to cooperate (by providing plans) for the achievement of the current goal.

Intentions may be either *active* or *suspended*. They are suspended when the execution of their topmost plan caused the generation of a subgoal, and the retrieval of relevant plans for that subgoal is not yet completed.

Coo-BDI engine. The engine for Coo-BDI differs from classical BDI interpreters, and is characterised by four macro-steps: (1) processing the mailbox; (2) processing the event queue; (3) processing suspended intentions; (4) processing active intentions. Before describing these four steps, we need to explain the mechanism for retrieving relevant plans, which is involved in steps 1 and 2 above. This mechanism starts when a new event enters the event queue. To keep the management of internal and external events homogeneous, we assume that, as soon as an external event enters the event queue, a new, empty intention is associated with it. The plan retrieval mechanism for a given triggering event Te consists of four sequential steps:

(a) The intention associated with Te is suspended.
(b) The local relevant plans for Te are generated and associated with the intention.
(c) According to the cooperation strategy for events matching Te, the set S of the agents expected to cooperate for handling the event is defined.
(d) If $S \neq \{\}$, a plan request for the event Te is created and sent to all the agents in S.

Now we can describe in more detail the four steps of the Coo-BDI engine.

1. Processing the mailbox: when an agent receives a plan request from another agent Ag, it sends to Ag its local plans which are relevant for that event and can be shared with Ag. On the other hand, when an agent receives an answer to a request for plans handling a certain event, it checks if the answer is still valid and if so it updates the intention associated with that event to include the plan just obtained.
2. Processing the event queue: after an event has been selected, the mechanism for retrieving plans triggered by that event is started.
3. Processing suspended intentions: the management of suspended intentions consists of looking for all suspended intentions which can be resumed. When an intention is resumed, the set of applicable plan instances is generated from the set of relevant plans associated with the intention, one applicable plan instance is selected and pushed on top of the corresponding intention stack. If the set of applicable plans is empty, the event for which plans had been collected cannot be handled and the corresponding intention is destroyed. The plans retrieved from cooperating agents may be discarded, added to the plan library, or used to replace local plans with a unifying triggering event, according to the acquisition policy related to the triggering event.
4. Processing active intentions: active intentions are processed as originally in the BDI architecture.

Finally, we point out that Coo-BDI (and, consequently, Coo-AgentSpeak) currently assumes that agents share the ontology used for writing the exchanged plans. This is a reasonable assumption for scenarios such as the one described in Section 4, but we expect to drop this assumption in future work, as discussed in Section 6. Note that we also assume that all plans are written in AgentSpeak: interoperability is not presently a concern of this work, although this too could be addressed in the future.

3. About *Jason*

A recent development in the practical aspects of AgentSpeak is the first release of ***Jason*** [3], an interpreter for an extended version of AgentSpeak(L), which allows agents to be distributed over the net through the use of SACI [8]. ***Jason*** is available *Open Source* under GNU LGPL at `http://jason.sourceforge.net`. It implements the operational semantics of AgentSpeak(L) as given in [4]. It also implements the extension of that operational semantics to account for speech-act based communication among AgentSpeak agents, as proposed in [9]. This, together with other mechanisms available in ***Jason***, allows us to implement in practice the ideas of plan sharing described in the previous section.

Besides interpreting the original AgentSpeak(L) language, some of the features available in ***Jason***, which are relevant for this work are:

- speech-act based inter-agent communication (and belief annotations on information sources);
- annotations on plan labels, which can be used by elaborate (e.g., decision theoretic) selection functions;
- the possibility to run a multi-agent system distributed over a network (using SACI);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- straightforward extensibility by user-defined internal actions, which are programmed in Java;

The idea of *internal actions* was introduced in [2]. They are actions (procedures) that are run internally by the agent; that is, they do not change the environment, thus they can be executed immediately without requiring an extra interpretation cycle. In fact, they can appear both in the context and in the body of a plan. Syntactically, they are differentiated from other actions by having a character '`.`' somewhere in the action symbol (i.e., the action name). The atom to the left of '`.`' is a library name, and to the right the name of an action within that library. This can be useful for users to organise (in separate libraries) the internal actions they create themselves (in ***Jason***, this is done in Java). An empty library

name (i.e., an action name starting with '`.`') makes reference to the standard library, which is provided with ***Jason***. One particularly important internal action available in the standard library is the one used for inter-agent communication: `.send(Ag,IlF,L)`, where `Ag` is an agent's name (or a set of names for multicast), `IlF` is the illocutionary force, and `L` is a literal in ***Jason***'s notation [3].

One of the illocutionary forces proposed in [9] for AgentSpeak inter-agent communication is $TellHow$. That paper gives a precise formalisation of this illocutionary force, which is used when the sender (s) intends the receiver (r) to add the content of the message to its plan library, rather than belief base. Suppose agent s executes the internal action "`.send(`r`,` $TellHow$`,P)`" while executing a plan, where `P` is, e.g., a logical variable instantiated with a string that can be parsed into an AgentSpeak plan (in fact, we here consider a *set of plans*, rather than exactly one). Then, when r receives that message, after checking whether s is a trusted information source, the plan parsed from the content of the message will be added to r's plan library. This provides the first essential requirement for allowing the sophisticated plan sharing approach of Coo-BDI to be implemented in practice using ***Jason***.

A second requirement is the availability of some mechanism for associating a plan with a list of properties; in this case, the list of plan access specifiers, for example, needs to be explicitly associated with plans. Fortunately, ***Jason*** implements a mechanism which makes the implementation of such specifiers straightforward. In ***Jason***, plans have labels, as proposed in [2], but instead of labels being simply atoms, they can be any predicate with annotations. Predicate "annotations" were introduced in [9], to be used in the agent's belief base for recording the sources of information (given that inter-agent communication for AgentSpeak was introduced in that paper). This extended syntax for predicates is as follows: $ps(t_1, \ldots, t_n)[a_1, \ldots, a_m]$, where ps is a predicate symbol of arity n, $t_1, \ldots, t_n$ are terms, and the m annotations $a_1, \ldots, a_m$ are also terms, all of which must be ground. Although originally defined to be used in predicates in the belief base, ***Jason*** also uses these extended predicates in plan labels. Plan labels appear to the left of a special symbol '`->`', and the remainder of a plan is as usual in AgentSpeak. Although a plan label can be any predicate with annotations, it is suggested that users write labels as a predicate of arity 0 (i.e., an atom) with annotations when necessary. So, for example, typical plan labels (for an arbitrary plan) would be:

```
aPlanLabel -> +b(X) : c(t) <- a(X).

anotherLabel[chanceSuccess(0.7),
    usualPayoff(0.9), anyOtherProperty] ->
  +b(X) : c(t) <- a(X).
```

It is then up to the user-defined selection functions to use such information in a plan's label according to the particular requirements of the given application. In ***Jason***, the interpreter is implemented in Java, and customisations to various functions used in the interpreter can be done by the user by overriding the methods of the `Agent` class that implement the standard versions of those functions. For example, to select an intended means from a set of applicable plans using application-specific plan properties, the programmer can override the following method:

```
public Option selectOption(List optList) {
}
```

which is called in the AgentSpeak interpretation cycle whenever a list of relevant and applicable plans[1] are obtained for the event being handled at that reasoning cycle. The default option selection function simply returns the first option in the Java `List` passed on parameter (the options are inserted in the list in the order the plans appeared in the AgentSpeak code).

Note that plan properties annotated in the plan label are copied when instances of a plan are placed in the set of intentions. Because of that, plan properties can be dynamically changed by programmers (during intention execution time) by executing user-defined internal actions that use certain Java methods available in ***Jason***. This provides a very interesting mechanism for the implementation of sophisticated selection functions, yet maintaining a neat notation in the agent programs.

Together with `selectOption`, the `selectEvent` and `selectIntention` customisable methods in ***Jason*** correspond to the three selection functions that are assumed as given in the AgentSpeak abstract interpreter [11, 2]. Besides these, another customisable function is `selectMessage`, which was introduced in [9]. It can be overridden when programmers need to customise the selection of the message in the agent's "mailbox" that is to be processed in the current reasoning cycle (the default function just chooses the first in the queue).

One final characteristic of ***Jason*** that is relevant here is the configuration option on what to do in case there is no applicable plan for a relevant event. If an event is relevant, it means that there are plans in the agent's plan library for handling that particular event (meaning that handling that event is normally a desire of that agent). If it happens that none of those plans are applicable at a certain time, this can be a problem as the agent does not know how to handle

1 In AgentSpeak, the notion of *option* refers specifically to the (viable) alternative courses of actions an agent knows in order to handle an event (e.g., achieve a goal). More specifically, the "options" are a set of plans that are relevant (for the event that was selected) and also *applicable* (i.e., the context of the plan is a logical consequence of the agent's belief base) at that moment in time. The particular course of action to which the agent commits itself is then referred to as the "intended means" for handling that event.

the event under the given circumstances. Ancona and Mascardi [1] discussed how this problem is handled in various agent-oriented programming languages. In ***Jason***, a configuration option is given to users, which can be set in the file where the various agents and the environment composing a multi-agent system are specified. The option allows the user to state, for events which have relevant but not applicable plans, whether the interpreter should discard that event altogether (`events=discard`) or insert the event back at the end of the event queue (`events=requeue`), so that it can be considered again later on.

Although Coo-AgentSpeak represents a significant improvement to the functionality of AgentSpeak agents, all the necessary mechanisms to implement Coo-AgentSpeak can be incorporated into ***Jason*** by using its extensibility or previously added general-purpose language constructs. The only modification to the interpreter that was required for ***Jason*** to cope with Coo-AgentSpeak was a third configuration option that is available to the users. When Coo-AgentSpeak is to be used, the option `events=retrieve` must be used in the configuration file. This makes ***Jason*** call the user-defined `selectOption` function *even when no applicable plans exist for an event*. This way, part of the Coo-BDI approach can be implemented by providing a special `selectOption` function which takes care of retrieving plans externally, whenever appropriate. Only in case this also fails, the existing plan failure mechanism available in ***Jason*** is put in place.

4. An Example Scenario

The scenario we shall use in the remainder of this paper has its roots in an application area that has recently become very popular: the use of Personal Digital Assistants (PDAs) for guided visits to museums, galleries, or cities.

A PDA used in that context should deliver content, in different media such as video, sound, and image, according to the museum room where it is currently located. However, because of their limited hardware resources, PDAs may not be able to keep internally the code for all the appropriate media players required during a visit, and they certainly cannot keep all the needed multimedia data. They need to cooperate with other PDAs and software/media providers to ensure the best guidance is given to the visitor, according to his/her preferences. When the PDA agent downloads movies or other media, it keeps them in memory only during the time it takes to show them to the user, and then discards them. Thus, the PDA will never have the movies themselves permanently in its memory, and it makes sense to configure it to always download movies rather than keep them internally, due to resource limitations.

Features such as reacting to the location in the museum, autonomously choosing the content to be shown to the user, and achieving the user's specific goals, make the use of BDI-based languages very suitable for modelling such PDA agents. The main obstacle to the adoption of a BDI approach is that existing BDI systems do not support any form of resource sharing. For the PDA technology to be really effective, dynamic downloading of code and content is a fundamental aspect. A more flexible BDI approach, in particular allowing the retrieval of plans as they become necessary, could solve the problem of the limited computational resources of a PDA for such applications, with the advantage of a high-level approach for the modelling of more autonomous devices.

In the examples we use in the next section, PDAs contain an agent which refers to `codeProvider` and `dataProvider` agents to download code for playing various media, and multimedia content, respectively. Of course, `codeProvider` and `dataProvider` agents reside on servers situated in appropriate points in the museum. At the start of the visit, PDA agents contain the plans that are needed for cooperating with the `codeProvider` and `dataProvider` agents, and for adapting possible museum tours to the user's preferences. During the visit, they temporary acquire plans for playing particular media, and, finally, at the end of the visit the PDA agents are reset, thus losing all acquired plans.

5. Coo-AgentSpeak

In order to achieve the goal of building an interpreter for AgentSpeak extended with the cooperation mechanisms supported by Coo-BDI, we exploit the features offered by ***Jason*** which are most suitable for the implementation of cooperative plan exchange. There are four main tasks to be accomplished:

1. in order to define the *cooperation strategy* of Coo-AgentSpeak agents, we need to include three more "given" functions[2] to the specification of each agent, in the same way that `trust` and `power` functions were added to agent specifications in [9];
2. including access specifiers and sources to plans;
3. keeping track of the events associated with intentions that were suspended because the agent needs to wait for an external plan retrieval to finish (i.e., that some other agent sends the plans that are required for carrying further the execution of that intention);

2 These functions are "given" in the sense that they are part of the interpreter of AgentSpeak yet are normally provided by the programmer, unless the default ones happen to be suitable; however, any sophisticated agent is likely to need application-specific versions of these functions.

4. customising ***Jason***'s interpreter to make the mechanism for retrieving external relevant plans transparent to the developer, as it is in Coo-BDI.

In the remainder of this section we show in more detail how the tasks above can be accomplished. In order to make the presentation clearer, we shall use the PDA application described in the previous section as a running example.

5.1. Cooperation strategy

The cooperation strategy can be easily expressed in Coo-AgentSpeak as a set of "system" (i.e., reserved) predicates in the belief base. The system predicates are:

- `cooAS_planSources`(*Te*,[Id_1,...,Id_n]),
- `cooAS_retrievalPolicy`(*Te*, *Retrieval*), and
- `cooAS_acquisitionPolicy`(*Te*, *Acquisition*),

corresponding to those introduced in Section 2.

A PDA agent which respects the requirements expressed in Section 4 could be configured with the following Coo-AgentSpeak cooperation strategy:

- ```
  cooAS_planSources("+!playMovie(Movie)",
                    [codeProvider]).
  cooAS_planSources("+!playSound(Sound)",
                    [codeProvider]).
  cooAS_planSources("+!retrieveData(Data)",
                    [dataProvider]).
  ```
- ```
  cooAS_retrievalPolicy("+!playMovie(Movie)",
                        noLocal).
  cooAS_retrievalPolicy("+!playSound(Sound)",
                        noLocal).
  cooAS_retrievalPolicy("+!retrieveData(Data)",
                        always).
  ```
- ```
  cooAS_acquisitionPolicy("+!playMovie(Movie)",
                          add).
  cooAS_acquisitionPolicy("+!playSound(Sound)",
                          replace).
  cooAS_acquisitionPolicy(
              "+!retrieveData(Data)", discard).
  ```

Note the difference in the acquisition policy for playing movies and sounds. We are assuming that a single application is used for playing movies, so once the corresponding plan has been downloaded, it will be present in the PDA agent's plan library until the end of the visit (hence the `add` policy). On the contrary, we assume that the `dataProvider` agent contains audio files with different formats each one needing a specific application which, due to memory limitation, cannot coexist in the PDA. For this reason a `replace` strategy is used to acquire plans for playing audio.

5.2. Plans

Jason already provides the means for including arbitrary annotations in plan labels; Coo-AgentSpeak can adopt ***Jason***'s notation for plans without requiring any further extension. We can take advantage of the "plan label annotation" feature to specify in all plan labels a `cooAS` structure of arity one; the argument of `cooAS` is a list which contains (at least for the time being) two terms, `accSpec` and `source`, both of arity one.

The argument of `accSpec` may range over {`private`, `public`, `only(`*`TrustedAgentsSet`*`)`}, while the argument of `source` may range over {`self`,id}. The meaning of these atoms is as explained in Section 2. Thus, for the PDA application, we can have Coo-AgentSpeak plans such as the following one:

```
p1[cooAS([accSpec(public),source(self)])]->
 +!playMovie(Movie)
 : moviePlayerInstalled(MoviePlayerCodeRef)
   <- start(MoviePlayerCodeRef, Movie).
```

The example above shows a plan that belongs to the `codeProvider` agent. It is a public plan (term `accSpec(public)`), owned by the `codeProvider` agent itself (term `source(self)`), and is used to play a movie (the triggering event is `+!playMovie(Movie)`). In case a movie player is believed to be installed (plan context `moviePlayerInstalled(MoviePlayerCodeRef)`) it is sufficient to start the movie player executable code with the movie as its argument (plan body `start(MoviePlayerCodeRef, Movie)`).

Other similar plans must be defined with the appropriate courses of action for the PDA agent to take under other circumstance, e.g., in case the required media player is not installed.

5.3. Intentions

We need to keep track of which intentions are waiting for the arrival of a certain plan from another agent. The simplest way to implement this extension in ***Jason*** is to use an *ad hoc* structure which is kept in the belief base. The advantage of having this information in the belief base is that agents can change the Coo-AgentSpeak information dynamically (e.g., by executing special user-defined internal actions from intended plan instances).

We associate an intention (which is explicitly represented in an event, so we will use events as references to the related intentions, for our convenience) with an external-plan request using system beliefs of the following form:
`cooAS_suspendedInt(`*`MsgID, Ev, Ags, Plans`*`)`.
Beliefs of this type will be used for updating the set of collected external plans *`Plans`* for the intention identified within *`Ev`*, when the agent receives a message in reply to the plan request with SACI identifier *`MsgId`*; *`Ags`* is initialised with the set of all agents to which the request was sent.
```

5.4. Engine

We now discuss the Coo-AgentSpeak engine, according to the following macro-steps defined in Coo-BDI.

1. **Processing the mailbox:** there are two types of messages which must be processed before any other: the requests for plans, and the answers to these requests. The `selectMessage` method provided by ***Jason*** is customised so that it gives priority to these types of messages; between the two, precedence should be given to the answers received for the agent's plan requests.

(a) Once a message of type ⟨`achieve, Ag, cooAS_sendPlansFor(Te)`⟩ is selected for being handled, the special event `+!cooAS_sendPlansFor(Te, Ag)` is included in the event queue of the receiver. The way this event is handled is explained in item 2 below (*processing the event queue*).

(b) Once a message of type ⟨`tellHow, Ag, P`⟩ in reply to a message identified by `MsgID` is selected for being processed, the information on the suspended intention associated with that `MsgID`, as registered in a predicate `cooAS_suspendedInt`, is updated in the belief base. Agent `Ag` is removed from the set `Ags` of agents that are still expected to reply, and all plans in `P` are added to `Plans` in that predicate. In case all agents have replied (i.e., `Ags` is empty), the intention can be resumed, by selecting one plan from `Plans`; before that, the library is updated according to the `acquisitionPolicy` specified in the triggering events of all received plans.

2. **Processing the event queue:** in this step we must consider two situations.

(a) A special event `+!cooAS_sendPlansFor`(*Te, Ag*) generated by the reception of a plan request is selected. The way these special events are generated was explained above. The `selectEvent` method provided by ***Jason*** can be customised so that it always selects `+!cooAS_sendPlansFor`(*Te, Ag*) events first (only `!startExternalPlanRetrieval`(*Te*) has higher priority)[3]. A system plan is provided by Coo-AgentSpeak with triggering event `+!cooAS_sendPlansFor(Te, Ag)`; it searches for all relevant plans for `Te` (this is accomplished by a special internal action implemented specifically for this purpose, making use of the unification algorithm used in ***Jason*** to check which plans in the plan library are relevant) and executes a `.send(Ag, tellHow, P)` action, where `P` is the set of retrieved plans.

(b) If a normal event `Te` is selected, two cases may occur. If the retrieval strategy for `Te` is `noLocal` and either `Te` has (locally) relevant and applicable plans, or there are no trusted agent for `Te`, then no request for external plans is issued and the computation continues in the same way as in the implementation of AgentSpeak. Otherwise, external plans need to be retrieved. In Coo-AgentSpeak, the `selectOption` method provided by ***Jason*** is customised so that, in this situation, it generates a new internal event `!startExternalPlanRetrieval(Te)`. The `selectEvent` method provided by ***Jason*** is customised so that it always selects `!startExternalPlanRetrieval(Te)` events first. Also, a system plan is provided (to be included in the agents' plan libraries) whose triggering event is the goal addition `+!startExternalPlanRetrieval(Te)` and the context has `cooAS_planSources(Te,AgList)`, and the plan body executes the `.send(AgList, achieve, cooAS_sendPlansFor(Te))` action (where `cooAS_planSources` is as defined above).

3. **Processing suspended intentions:** in Coo-AgentSpeak, this step is performed together with the handling of messages, as described above. Nothing special is required for this step of Coo-BDI when implemented in ***Jason***.

4. **Processing active intentions:** this step is the same as originally defined in AgentSpeak. See rules **Action**, **Achieve**, **Test$_1$**, **Test$_2$**, **AddBel**, and **DelBel** in [9, appendix].

Finally, we show how steps 1 and 2 above work in our PDA example. Let us start from the situation where a PDA agent `pda1` selects a normal event with triggering event T =`+!playMovie(movie1)`, and let us assume that `pda1` has no relevant plans for T. According to the strategy specified in the example of Section 5.1, agent `codeProvider` is the only known source of plans for T, therefore the intention that generated T is suspended and a new internal event is created; an empty intention is associated to the event and its triggering event is as follows:

```
!startExternalPlanRetrieval(
  "+!playMovie(movie1)")
```

Then the internal goal is selected from the event queue and the only relevant plan that is found is the system plan specifically designed for handling this event. An instance of that plan is obtained by instantiating the variables `Te` and `AgList` in its triggering event with `playMovie(movie1)` and `[codeProvider]`, respectively.

3 Note that this is the behaviour of the default `selectMessage` and `selectEvent` functions provided by Coo-AgentSpeak. As the methods implementing these and other selection functions can be overridden at the individual agent level, the user can further customise these methods, e.g., to give the agent more autonomy so as to only help other agents by sending plans when it can afford to do so.

Then the action `.send([codeProvider],achieve, cooAS_sendPlansFor(playMovie(movie1)))` is executed and the `codeProvider` agent receives, in its mailbox, the following message:

⟨`achieve,pda1, cooAS_sendPlansFor(playMovie(movie1))`⟩.

The message is processed by generating an event for:

T' =`+!cooAS_sendPlansFor( playMovie(movie1),pda1)`

which is, then, selected; the only relevant (and applicable) plan for agent `codeProvider` whose triggering event unifies with T' is the system plan designed for handling special events such as T'. Its execution collects all plans available in agent `codeProvider` whose triggering events unify with `playMovie(movie1)`; let *P* be such set of plans. The `codeProvider` agent then performs the action `.send(pda1,tellHow,`*P*`)`. Subsequently, `pda1` selects, from its mailbox, the message <`tellHow,codeProvider,`*P*> and, finally, it can permanently add *P* to its plan library, resume the suspended intention for `playMovie(movie1)` (since no other agent is expected to send further plans), and play `movie1` to the museum visitor.

6. Conclusions

In this paper, we have shown how Coo-BDI, a sophisticated mechanism for plan exchange among BDI agents, can be applied to AgentSpeak, and made practical by using the ***Jason*** interpreter. The ability to exchange plans is essential for the development of multi-agent systems where agents are reactive planning systems. It allows agents to change their know-how over time, accounting both for adaptation as well as boundedness of computational resources (e.g., to avoid keeping large numbers of plans in plan libraries). To the best of our knowledge, this is the first such mechanism put in place for an agent-oriented programming language. We expect the Coo-AgentSpeak extensions to be available with ***Jason*** (also as *Open Source*) in the near future.

Future work should address the development of large scale multi-agent systems, so that we can further assess the adequacy of our ideas in such context. There is ongoing work on AgentSpeak to incorporate the use of ontologies; with that, we can drop the assumption of shared ontology among all agents, yet agents can make sure there is no ontological discrepancies in the plans they exchange. We also plan to formalise the ideas presented here as part of the operational semantics of AgentSpeak. Finally, by using SACI "yellow pages" functionalities, we can improve the mechanism for retrieving the necessary plans so that agents do not need to worry which agents to ask for the plans they need. We believe that with the future development of this approach, it will allow the implementation of sophisticated teams of autonomous agents, having the advantage of agents being coded in an agent-oriented programming language that has formal semantics and is inspired by the well known BDI architecture for cognitive agents.

References

[1] D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In *Proc. of DALT-03*, 2003.

[2] R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. Johnson, editors, *Proc. of AAMAS-02*, pages 1294–1302. ACM Press, 2002.

[3] R. H. Bordini, J. F. Hübner, et al. ***Jason****: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, first release edition, 2004. `http://jason.sourceforge.net/`.

[4] R. H. Bordini and A. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3), 2004. To appear.

[5] M. Dastani, B. van Riemsdijk, F. D. Frank, and J.-J. C. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Proc. of ProMAS-03*, 2003.

[6] G. de Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog: A concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

[7] M. Fisher, C. Ghidini, and B. Hirsch. Organising logic-based agents. In *Proceedings of FAABS-02*, number 2699 in LNCS, pages 15–27. Springer-Verlag, 2003.

[8] J. F. Hübner. *SACI — Simple Agent Communication Infrastructure*, 2003. Programming Manual. `http://www.lti.pcs.usp.br/saci/`.

[9] A. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proc. of DALT-03*, 2003.

[10] D. V. Pynadath and M. Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems*, 7(1–2):71–100, 2003.

[11] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van de Velde and J. Perram, editors, *Proc. of MAAMAW-96*, number 1038 in LNAI, pages 42–55. Springer-Verlag, 1996.

[12] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[13] B. van Riemsdijk, W. van der Hoek, and J.-J. C. Meyer. Agent programming in Dribble: from beliefs to goals using plans. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proc. of AAMAS-03*, pages 393–400. ACM Press, 2003.

[14] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. on Software Engineering and Methodology*, 12(3):317–370, 2003.