

# Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange

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

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

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

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 AgentSpeak(L). Jason also provides mechanisms that allow the specification of plan permissions, which are important in the cooperation context. This paper shows how elaborate plan exchange can take place between AgentSpeak agents implemented with Jason. It also discusses an application in which plan sharing is essential.*

## 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 focused mainly on societal aspects of building multi-agent systems, such as groups and roles, taking for granted the development of individual agents and the cognitive aspects of such agents. 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 [6]. 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 necessary means for autonomous agents to exchange plans, so that agents can increase their know-how 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 [7], and ConGolog [5].

Our start 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 a BDI agent-oriented programming language called AgentSpeak(L), in the context of *Jason* [3], an interpreter for that language recently made available. AgentSpeak(L) was 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 the communication of 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.

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); only the features that are relevant for the work here are discussed. Then, in Section 4 we described 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, galleries, etc. That scenario is used in Section 5, where we present Coo-AgentSpeak, the instantiation of Coo-BDI for AgentSpeak(L) in particular, and discuss how these ideas can be made practical in *Jason* with just minor changes. The paper assumes that the reader is familiar with the BDI architecture and also with the AgentSpeak(L) language.

## 2. Coo-BDI: Cooperation in BDI Agent Languages

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.

1. **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).
2. **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”, which correspond to Coo-BDI “desires”, 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 been modified in order to comply with the syntax 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 or clarity.

*Cooperation strategy.* The cooperation strategy of an agent *Ag* includes the set of agents with which it is expected to cooperate, 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 functions:

- *trusted*(*Te*, *TrustedAgentSet*), where *Te* is a (not necessarily ground) trigger and *TrustedAgentSet* is the set of agents that *Ag* will contact in order to obtain plans relevant for *Te*.
- *retrievalPolicy*(*Te*, *Retrieval*), where *Retrieval* may assume the values *always* and *noLocal*, meaning that relevant plans for the trigger *Te* must be retrieved from other agents in any case, 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 used and discarded, or added to the plan library, or used to update the plan library by replacing all the plans triggered by *Te*.

*Plans.* Besides the standard components which constitute BDI plans, Coo-BDI plans also have a *source* which determines the first owner of the plan, 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 possesses the plan) and *Ag* (the agent was originally from *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 contained in *TrustedAgentSet*).

*Intentions.* Intentions are characterised by “standard” components plus components introduced to manage the external plan retrieval mechanism:

- the relevant plans which have been already collected for the management of the current goal; and
- the set of identifiers of those agents which are still expected to cooperate for the management of the current goal.

Intentions may be either *active* or *suspended*. They are suspended when the management 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 departs from the classical BDI one, 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 consists of four sequential steps:

- (a) The intention associated with the event ( $Te$ ) is suspended.
- (b) The local relevant plans for the event are generated and associated with the intention.
- (c) According to the cooperation strategy, 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  $Te$  (i.e., a message with `achieve` as illocutionary force and `sendPlansFor( $Te$ )` as propositional content) 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 an `achieve` message with `sendPlansFor( $Te$ )` content from another agent  $Ag$ , it sends to  $Ag$  its local plans which are both relevant for that event and sharable with  $Ag$ . On the other hand, when an agent receives an answer to a request for plans handling a certain event (`tellHow` message with a plan  $P$  as content), it checks if the answer is still valid and if so it updates the intention associated with the 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

managed and the corresponding intention is destroyed. The plans retrieved from cooperating agents may be discarded, or 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 managed as in the BDI architecture.

### 3. About Jason

A recent development in the practical aspects of AgentSpeak is the first release of *Jason*, 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> [3]. 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, and other mechanisms available in *Jason* allow 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 feature 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 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 `.send(ag, ilf, l)`, where *ag* is an agent’s name, *ilf* is the illocutionary force, and *l* is a literal in **Jason**’s notation [3].

Of particular interest are the first two items. One of the illocutionary forces proposed in [9] for AgentSpeak inter-agent communication is *TellHow*. That papers give 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 following internal action while executing a plan:

```
.send(r, TellHow, P)
```

where *P* is, e.g., a logical variable instantiated with a string that can be parsed into an AgentSpeak(L) plan. 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 plan with a list of properties; in this case, the list of plan access specifiers, for example, need 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” was 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:  $ps(t_1, \dots, t_n)[a_1, \dots, a_m]$ , where *ps* is a predicate symbol of arity *n*,  $t_1, \dots, t_n$  are terms, and the *m* annotations  $a_1, \dots, a_m$  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. 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, if necessary. So, for example, typical plan labels (with an arbitrary plan) would be:

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

anotherLabel[chanceSuccess(.7),
usualPayoff(.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 parts of the interpreter can be done by the user by redefining standard methods in the classes that read the AgentSpeak source code; these Java classes are automatically generated by **Jason**. In particular, for using plan properties in ways that are application-specific, the programmer can redefine the following method:

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

which is called in the AgentSpeak(L) interpretation cycle whenever a list of relevant and applicable plans<sup>1</sup> 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 plan appear in the AgentSpeak(L) code).

Note that the 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 by executing certain Java methods from within an internal action. 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 is used for programmers to customise the choice of a message from the agent’s “mail box” 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 (representing 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 the situation at that time. Ancona and Mascardi [1] discussed how this problem is handled in various

<sup>1</sup> In BDI terminology, an “option” is one of the alternative courses of actions an agent know to achieve a desire (i.e., a goal). The one particular course of action it commits to is referred to as the “intended means” for achieving that desires; at that point the chosen course of action is intended by the agent. In AgentSpeak(L), the idea of an “option” translates into the set of plans that are relevant (for the event that was selected) and also *applicable* at that moment in time.

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`). One of the few modifications that were required for *Jason* to cope with *Coo-AgentSpeak* (which will be presented in the next section) 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.

#### 4. A Sample 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 adoption of Personal Digital Assistants (PDAs) for visiting 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 codes 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 that is needed to show them to the user, and then discards them. Thus, the PDA will never have the movies themselves permanently in its internal disk, 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 the media player, and multimedia data, 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 temporarily acquire plans for playing particular media, and, finally, at the end of the visit the PDA agents are reset and loose all the acquired plans.

#### 5. *Coo-AgentSpeak*: Extending *AgentSpeak* with Cooperation

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<sup>2</sup> 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 it needs to carry further that intention);
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 will use a running example based on the PDA application described in the previous section.

<sup>2</sup> 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 can be used; any sophisticated agent is likely to need application-specific versions of these functions.

## 5.1. Cooperation strategy

The cooperation strategy can be easily expressed in Coo-AgentSpeak as a set of “system” (i.e., reserved) beliefs in the belief base. These beliefs define three functions:

- `cooAS_planSources(Te, [Id1, ..., Idn]),`
- `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 just a single application is needed for playing movies, hence once the corresponding plan has been downloaded, it will be present in the PDA agent’s plan library until the end of the visit (add policy). On the contrary, we assume that the dataProvider agent contains audio files with different formats each one needing a specific application which, for memory space limits, cannot coexist in the same PDA. For this reason a replace strategy is used to acquire plans for playing sound.

## 5.2. Plans

*Jason* already provides the means for including arbitrary annotations in plan labels; Coo-AgentSpeak can adopt *Jason*’s syntax for plans without requiring any further extension. We can take advantage of *Jason* “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:

```
pl[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 in case the media players are not installed, for the PDA to start playing a movie, and so on.

## 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., but the execution of special user-defined internal actions in the agent’s plans).

We represent the relation among intentions (which are unequivocally related to events, so we will use events as identifiers for our convenience) and an external plan request using the system beliefs of the following form:

```
cooAS_suspendedIntention(MsgID, Ev).
```

The information contained in this relation will be used when the `selectMessage` function of the agent receives a message in reply to the message sent with a request for a plan. The unique SACI message identifier (`MsgID`) is registered, so that the intention that was suspended in connection with that plan request can be activated again.

## 5.4. Engine

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

1. **Processing the mailbox:** there are two types of messages which must be processed before any other. They are the requests for plans, and the answers to these requests. The `selectMessage` method provided by **Jason** is customised so that it gives priority to types of messages; between the two, precedence should be given to the answers received for the agent's plan requests<sup>3</sup>.

- (a) Once a message of type  $\langle \text{achieve}, \text{Ag}, \text{cooAS\_sendPlansFor}(Te) \rangle$  is selected for being handled, the special event  $+\text{!cooAS\_sendPlansFor}(Te, \text{Ag})$  enters the event queue of the receiver. The way this event is managed as explained in item 2 below (*processing the event queue*).
- (b) Once a message of type  $\langle \text{tellHow}, \text{Ag}, P \rangle$  is selected for being processed, it is possible to activate again the suspended intentions that were waiting for that message (identified by a unique *MsgID*), and the plan library can be updated according to the `acquisitionPolicy` of the triggering event of that plan. The information on which intention is suspended, waiting for *MsgID*, is recorded in the special **Coo-AgentSpeak** system predicate `cooAS_suspendedIntention`.

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

- (a) A special event  $+\text{!cooAS\_sendPlansFor}(Te, \text{Ag})$  generated by the reception of a plan request is selected. The way these special events are generated were explained above. The `selectEvent` method provided by **Jason** can be customised so that it always selects  $+\text{!cooAS\_sendPlansFor}(Te, \text{Ag})$  events first (only `!startExternalPlanRetrieval(Te)` has higher priority). A system plan is provided by **Coo-AgentSpeak** whose triggering event is  $+\text{!cooAS\_sendPlansFor}(Te, \text{Ag})$  and whose body 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 for each relevant plan *P* retrieved.
- (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 computa-

tion 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 goal `!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 by **Coo-AgentSpeak** (to be included in the agents' plan libraries) whose triggering event is  $+\text{!startExternalPlanRetrieval}(Te)$  goal and whose context is `cooAS_planSources(Te, AgList)`, and the plan body calls the `.send(Ag, achieve, cooAS_sendPlansFor(Te))` action for each agent *Ag* in *AgList* (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 applied to **Jason**.
- 4. **Processing active intentions:** this step is the same as in **AgentSpeak**. See the rules **Action**, **Achieve**, **Test<sub>1</sub>**, **Test<sub>2</sub>**, **AddBel**, **DelBel** in the appendix of [9].

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 the normal event  $Te = +\text{!playMovie}(\text{movie1})$ , and let us assume that *PDA1* has no relevant plans for *Te*; according to the strategy specified in the example of Section 5.1, agent `codeProvider` is the only known source of plans for *Te*, therefore the intention for *Te* is suspended and the new internal goal

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

is generated. 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 substituting the variables *Te* and *AgList* in its triggering event with `playMovie(movie1)` and `[codeProvider]`, respectively. 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:

```
Te' = +!cooAS_sendPlansFor(
  playMovie(movie1), PDA1)
```

<sup>3</sup> Note that this is the behaviour of the `selectMessage` as provided by **Coo-AgentSpeak**. As this methods can be overridden at the individual agent level, the user can further customise these methods so as to give the agent more autonomy to only give priority to helping other agents by sending plans when it can afford to do so.

which is, then, selected; the only applicable plan for agent `codeProvider` whose triggering event unifies with  $Te'$  is the system plan designed for handling special events such as  $Te'$ . Its execution collects all plans available in agent `codeProvider` whose triggering events unify with `playMovie(movie1)`. If we assume that there exists just one such plan  $P$ , then `codeProvider` 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, activate the suspended intention for `playMovie(movie1)`, 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(L)*, and made practical by using an interpreter for it called *Jason*. The ability to exchange plans is essential for the development of multi-agent systems using agent-oriented programming languages. It allows agents to change their know-how over time, accounting both for adaptation as well as limited computational resources (e.g., when agents cannot keep a large number of plans at a time in their plan libraries). To the best of our knowledge, this is the first such mechanism put in place for an agent-oriented programming languages. 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. Also, 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, very sophisticated teams of autonomous agents can be implemented, having the advantage of agents being specified with an agent-oriented programming language that has formal semantics, and that 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 the Workshop on Declarative Agent Languages and Technologies (DALT-03)*, held with AAMAS-03, 15 July, 2003, Melbourne, Australia, 2003. To appear in Springer LNAI series.
- [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 *Proc. of the 1st Int. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, 15–19 July, Bologna, Italy, pages 1294–1302, New York, NY, 2002. ACM Press.
- [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, Jan 2004. <http://jason.sourceforge.net/>.
- [4] R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in *AgentSpeak(L)*. *Annals of Mathematics and Artificial Intelligence*, 2004. To appear.
- [5] 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.
- [6] M. Fisher, C. Ghidini, and B. Hirsch. Organising logic-based agents. In *Proc. of the 2nd NASA/IEEE Goddard Workshop on Formal Approaches to Agent Based Systems (FAABS 2002)*, Greenbelt, MD, October 29-31, LNCS 2699, pages 15–27, Berlin, 2003. Springer-Verlag.
- [7] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer. A programming language for cognitive agents: Goal directed 3apl. In *Proc. of the 1st Int. Workshop on Programming Multiagent Systems (ProMAS-03)*, held with AAMAS-03, 15 July, 2003, Melbourne, Australia, 2003. To appear in Springer’s LNAI series.
- [8] J. F. Hübner. *SACI — Simple Agent Communication Infrastructure*, 2003. Programming Manual. <http://www.lti.pcs.usp.br/saci/>.
- [9] Á. 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 the Workshop on Declarative Agent Languages and Technologies (DALT-03)*, held with AAMAS-03, 15 July, 2003, Melbourne, Australia, 2003. To appear in Springer’s LNAI series.
- [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 the 7th Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’96)*, 22–25 January, Eindhoven, The Netherlands, LNAI 1038, pages 42–55, London, 1996. Springer-Verlag.
- [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 *Proc. of the 2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-2003)*, Melbourne, Australia, 14–18 July, pages 393–400, New York, NY, 2003. ACM Press.
- [14] M. J. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.