Coo-BDI: Extending the BDI Model
with Cooperativity

Davide Ancona and Viviana Mascardi

DISI - Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy
{davide,mascardi}@disi.unige.it

Abstract. We extend the BDI architecture with the notion of cooperativity. Agents can cooperate by exchanging and sharing plans in a quite flexible way. As a main result Coo-BDI promotes adaptivity and sharing of resources; as a by product, it provides a better support for dealing with the situation when agents do not possess their own procedural knowledge for processing a given event.

1 Introduction

Intelligent agents [20, 9] are a powerful abstraction for conceptualizing and modeling complex systems in a clear and intuitive way. For this reason many declarative agent models have been proposed in the last years. One of the most successful is the Belief, Desire and Intention (BDI) one [17], whose wide appreciation is witnessed by the development of a BDI logic [18], the definition of BDI-based languages (AgentTalk1, 3APL [6], AgentSpeak(L) [16]) and the creation of BDI-based development tools such as JACK [4], PRS [12] and dMARS [7].

Despite this large consensus, however, many BDI concepts are not interpreted in a uniform way. To make an example, let us consider the notion of “goals”, namely, consistent sets of desires. For many authors, they should be explicitly represented in order to reason about their properties and ensure their consistence. This explicit representation is not available in current BDI systems and languages such as dMARS, AgentSpeak(L) and JACK where goals, if represented at all, have only a transient representation as a type of event. A similar situation takes place with events (or subgoals, since they are considered the same entity in implemented systems) which cannot be dealt with because of the lack of plans suitable for managing them. In the original specification of the BDI architecture and in many high level BDI languages this case is not considered at all, and implemented systems solve it in very different ways: in most of them, the event or subgoal is simply dropped; in other ones the definition of a default “catch-all” plan is required. In this paper we propose an extension to the basic BDI model, for

1. representing events, goals, and the relationships between them in a conceptually clear way, and

2. allowing agents to cooperate by exchanging plans to cope with the case
   no local applicable plans are available for managing an event (or goal, or
   subgoal).

The idea behind the first extension is to keep events and goals (that we will call
desires for uniformity with the Belief, Desires, Intentions acronym) separate.
The agent has an event queue containing events generated by the environment
or by other agents. Managing an event means removing it from the event queue
and generating the set of desires associated with the event. For example, the
event “reception of an e-mail asking to join friends for a drink at the Madeleine
Café at 7 p.m.” may generate the desires “cancel all the appointments for this
evening” and “reach the Madeleine Café at 7 p.m.”. Desires are maintained as
explicit information in an ad hoc structure.

Let us go on with the previous example, and let us suppose that the agent
desires to reach the Madeleine Café at 7 p.m., but it has no procedural knowledge
(namely, no plans) for achieving this desire. In the existing BDI languages and
systems, the agent would give up or would apply a default plan. In the extension
we propose, the agent asks to other agents if they have a good plan for reaching
the Madeleine Café. Some of the agents contacted for advice could cooperate by
suggesting plans for achieving this desire, allowing the requiring agent to reach
its friends.

Our extension allows to implement agents that dynamically change their
behavior by cooperating with other agents. This feature turns out to be useful
in many application fields such as:

Personal Digital Assistants (PDAs). The physical resources of a PDA are usually
limited and for the PDA technology to be really effective, dynamic linking of code
is a fundamental aspect. If the traditional BDI approach is adopted, modeling
dynamic linking is not easy: BDI agents are expected neither to possess the
capability of looking for external plans nor to dynamically extend their plan
library with the plans retrieved from other agents. Our extension can cope with
both issues and allows an agent to discard the external plans after their usage:
this may prove useful when the space resources of the PDA have strict bounds.

Self-repairing agents. By self-repairing agent we mean an agent situated in a
dynamically changing software environment and able to identify the portions of
its code that should be updated to ensure its correct functioning in the evolving
environment. When the agent finds out pieces of code that have become obsolete,
the agent replaces them with new code, provided by a server agent, without
needing to stop its activity and the activity of the whole system. Coo-BDI allows
to dynamically replace local plans with external ones and thus can be used for
modeling such a self-repairing agent.

Digital butlers. A “digital butler” is an agent that assists the user in some task
such as managing her/his agenda, filtering incoming mail, retrieving interesting
information from the web. A typical feature of a digital butler is its ability to
dynamically adapt its behavior to the user needs. This ability is achieved by
cooperating both with more experienced digital butlers and with the assisted
user. In our setting we may think that, through a user friendly interface, the
user may train the agent by showing the right sequence of actions to perform in particular situations. The agent may treat this sequence as an externally retrieved plan which enriches its plan library, as well as plans retrieved from peer agents.

More in general, our extension is suitable for modeling all kinds of “learning agents” which learn from the interaction with other agents.

The structure of the paper is the following: in Section 2 we recall the main BDI concepts and survey the common approaches to the “no applicable plans” problem; in Section 3 we introduce Coo-BDI, our BDI model extended with co-operativity among agents for sharing plans, and then provide its full specification. Section 4 concludes.

2 The BDI Model

The BDI model is characterized by the following concepts:
- **Beliefs**: the agent’s knowledge about the world.
- **Desires**: objectives to be accomplished; are similar to goals, but they do not need to be consistent.
- **Intentions**: plans currently under execution.
- **Plans**: “recipes” representing the procedural knowledge of the agent. They are usually characterized by a trigger which fires the adoption of the plan, a precondition that the current state must satisfy for the plan to be applicable, a body of actions to perform, an invariant condition that must hold during all the plan execution, a set of actions to be executed if the plan execution terminates successfully and a set of actions to be executed in case of plan failure. Plans are static: they do not change during the agent’s life cycle.

All BDI systems also include an event queue where external events (perceived from the environment) and internal subgoals (generated by the agent itself while trying to achieve a main goal) are stored.

The typical BDI execution cycle is characterized by the following steps:
1. observe the world and the agent’s internal state, and update the event queue consequently;
2. generate possible new plans whose trigger event matches an event in the event queue (“relevant” plans) and whose precondition is satisfied (“applicable” plans);
3. select one plan from this set of matching plans for execution;
4. push the selected plan onto an existing or new intention stack, according to whether or not the event is a (sub)goal;
5. select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

This description is not detailed enough to answer to the question: what should the agent do if the set of applicable plans is empty? This issue is usually not considered by the BDI high level specifications. In [17], Rao and Georgeff write that after the option generator reads the event queue and returns a list of options
(namely, a set of applicable plans), the deliberator selects a subset of the options to be adopted and adds these to the intention structure. We can guess that if no options (plans) can be adopted the interpreter simply jumps to the successive step but this is not explicitly established by the specification. In [7], d'Inverno, Kinny, Luck and Wooldridge provide a more detailed specification than the one discussed in [17], but nevertheless, the case that no applicable plans are found is not considered. In the specification of AgentSpeak(L) [16], the $S_O$ function selects an option or an applicable plan from a set of applicable plans $O_*$. Also in this case, if $O_*$ is empty, it is not clear what the interpreter does.

When we move from specification to implementation, the problem above cannot be ignored any more. There are different implementation solutions BDI systems adopt. When the implemented BDI systems AgentTalk, JACK, dMARS select an event from the event queue for which there are no applicable plans, they simply ignore the event and delete it from the event queue. The implementation of 3APL (http://www.cs.uu.nl/3apl/) adopts a default plan if no other plans can be applied to deal with the selected event. In the implementation of AgentSpeak(L) developed by R. Bordini et al. [1] (http://www.inf.ufrgs.br/”Massoc” there are two options of what to do with events which are relevant but not applicable: the user can either ask the interpreter to discard these events or to keep them in the set of events. In the latter case, if eventually a plan becomes applicable, then the agent can do something.

This short overview shows that both at the specification and implementation level the “no applicable plans” problem is solved in a rather ad hoc way. Our proposal, motivated by the applications sketched in the introduction, focuses on the agents ability of cooperating.

3 A Full Specification of Coo-BDI

3.1 Overview

Coo-BDI (Cooperative BDI) is based on the dMARS specification [7] and extends the traditional BDI architecture described in Section 2 in many respects. The first Coo-BDI extension is that external events and main desires are kept separate. In Coo-BDI there are two structures which maintain events and desires: the “event queue” which only contains external events, and the “desire set”, which only contains achieve desires generated by events. We distinguish between main desires, which are kept in the desire set, and subaltern desires, which are generated while trying to achieve a main desire and remain implicit in the intention stack structure. When a main desires fails backtracking is applied and a “fresh” (not already attempted) plan instance is selected for it. The main desire is removed by the set if there are no more fresh plan instances for it. Subaltern desires are not backtracked, both to keep the Coo-BDI interpreter and data structures simpler and to maintain the same strategy implemented by dMARS.

The main extension of Coo-BDI, however, involves the introduction of co-operations among agents to retrieve external plans for achieving desires (both
main and subaltern), the introduction of default plans, the extension of specific (non-default) plans with access specifiers, the extension of intentions to take into account the external plan instances retrieval mechanism and the modification of the Coo-BDI engine to cope with all these issues.

Cooperation strategy. The cooperation strategy (or, simply, the cooperation) of an agent $A$ includes the set of agents with which is expected to cooperate, the plan retrieval policy and the plan acquisition policy. The cooperation strategy may evolve during time allowing the maximum flexibility and autonomy of the agents.

Plans. Coo-BDI specific plans share with “classical” ones the trigger, the precondition, the body, the invariant and the two sets of success and failure actions. Besides these components, they also have an access specifier which determines the set of agents the plan can be shared with. It may assume three values: private (the plan cannot be shared), public (the plan can be shared with any agent) and only(TrustedAgents) (the plan can be shared only with the agents contained in the TrustedAgents set). Default plans are introduced to ensure that for each desire, at least one plan instance exists for its management. Default plans are private, otherwise they would be continuously exchanged between agents and this makes little sense.

Intentions. Coo-BDI intentions are in a one-to-one relation with main desires: each intention is created when a new main desire enters the desires set, and it is deleted when the main desire fails or is achieved. Intentions are characterized by “standard” components plus components introduced to manage the external plan retrieval mechanism. External plans are retrieved, according to the retrieval policy, both for main and for subaltern desires.

Coo-BDI engine. The engine of Coo-BDI departs from the classical one to take into account both desires generation and cooperations. It is characterized by three macro-steps:

1. process the event queue;
2. process suspended intentions;
3. process active intentions.

Before describing these three steps, we need to explain the mechanism for retrieving relevant plans, since such mechanism is essential for understanding how steps 1 and 3 work. 1) The intention is suspended. 2) The local relevant plans for the desire are generated and associated to the intention. 3) According to the cooperation, the set $S$ of the agents expected to cooperate is defined. 4) A plan request for the desire is created and it is sent to all the agents in $S$.

Events in the event queue may be either cooperation or ordinary events. Cooperation events include requests of relevant plan instances for a desire and answers to a plan request. Ordinary events include at least messages reception and notification of updates performed on the agent’s beliefs set. When an agent
receives a request of relevant plan instances for a desire, it looks for all the local relevant plan instances for the desire which can be shared with the requesting agent (the ones whose access specifier is public or includes the requesting agent) and sends them to it. On the other hand, when an agent receives an answer to a plan request for a desire, it checks if the answer is still valid and if so it updates the intention associated with the desire to include the just obtained plan instances and to remember that answer. Finally, if the event is an ordinary one, the set of corresponding desires is generated and added to the desire set. For each new desire an empty intention is created and the mechanism for retrieving relevant plans is started.

The management of suspended intentions consists in 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 plan instances except for the already failed instances, one applicable plan is selected and the corresponding plan instance is created. If the applicable plan instances set is empty, the desire fails and is deleted from the desire set, and the intention is destroyed. Otherwise, the selected plan instance is pushed onto the intention stack. If the selected plan is an externally retrieved one, it may be used and discarded, or added to the plan library or used to replace plans with a unifying trigger according to the acquisition policy.

Finally, the management of active intentions is similar to the one discussed in [7] apart from the management of a desire achievement which starts the mechanism for retrieving relevant plans in case the desire is not a logical consequence of the agent’s current beliefs.

3.2 Structural specification of Coo-BDI

The following subsections describe in detail the structure of the components of the Coo-BDI architecture.

Beliefs, desires, queries and actions. Beliefs are ground atoms (namely, symbols of predicates applied to terms, without variables). To keep the implementation of a Coo-BDI working interpreter as simple as possible, we avoid including negative atoms among beliefs.

Desires have the form achieve(Atom), where Atom is a positive atom.

Queries are denoted by query(SituationFormula) where a situation formula is either an atom, or the constant true or false, or conjunctions and disjunctions of situation formulas.

Actions may be internal or external. Internal actions are updates of the database of an agent’s beliefs: add(Atom) and remove(Atom). External actions include at least the ability of sending messages to agents: send(AgentId, Message).

Plans. Plans can be of two different forms: either specific or default. A specific plan is defined by the following components: an access specifier AccessSpecifier, a trigger Trigger, a precondition Precondition, a body Body, an invariant Invariant,
and two sets of internal actions *SuccessActions* and *FailureActions*. The access specifier may assume the following values:

- *private*, meaning that the plan cannot be provided;
- *public*, meaning that the plan can be provided to any agent;
- *only(TrustedAgents)* where *TrustedAgents* is a set of agent identifiers specifying the only trusted agents which can receive instances of the plan.

The trigger of a specific plan is the desire the plan is designed to achieve. Like queries, preconditions and invariants are situation formulas. Plan bodies are non empty trees where nodes are execution states and edges are labelled by either desires, or queries, or (both external and internal) actions. Desires in plan bodies are called *subaltern*.

Finally, success and failure actions are sequences of internal actions. A default plan is just a specific plan with no access specifier, no trigger and no precondition.

**Plan instances.** A plan instance is a pair (*Plan, Substitution*) formed by a plan and a substitution, that is a map from variables to terms (we refer to [19] for the notion of substitution and most general unifier).

A plan instance (*Plan, Substitution*) is called *relevant* w. r. t. a desire if either *Plan* is specific and *Substitution* is a most general unifier for that desire and the trigger of *Plan*, or *Plan* is a default plan and *Substitution* is the empty substitution.

A plan instance (*Plan, Substitution*) is called *applicable* if either *Plan* is specific and the formula obtained by applying *Substitution* to the precondition of *Plan* is a logical consequence of the beliefs of the agent’s data base, or *Plan* is a default plan and *Substitution* is the empty substitution.

The execution of a plan instance is defined by its plan instance *Instance* together with the computed substitution *Substitution*, the current state *CurrentState* of the body of the plan of *Instance*, the set *NextStates* of the remaining siblings of *CurrentState* which have not been executed yet.

**Intentions and cooperation requests.** An intention is composed by a stack *Stack* of executions of plan instances, a unique identifier *IntentionId*, a status *Status*, a set of plan instances *RelevantInstances*, a set of agent identifiers *WaitingOnAgents*, and a set of failed plan instances *FailedInstances*; furthermore a relation *IntentionRequest* associate with any intention identity its current (if any) request for cooperation, and with any request the intention it originates from, if the request is still valid.

The stack of executions of plan instances is similar in spirit to the execution stack of logic programs. The intention identifier is necessary to relate an intention to the desire it originated from; such a desire is called *main*.

The status may be either *suspended* or *active*. An intention is suspended if the execution of the plan instance on top of the stack needs to achieve a desire *Desire* for which no plan instance has been selected yet; in this case
RelevantInstances contains the relevant plan instances which have been already collected for Desire, WaitingOnAgents contains all the identifiers of those agents which are still expected to cooperate for achieving Desire, and IntentionRequest associates the intention identifier with the current cooperation request for Desire.

A cooperation request is specified by a unique identifier RequestId, the identifier AgentId of the requesting agent, and the desire Desire to achieve.

Events. There are two kinds of events: cooperation and ordinary events. A cooperation event is either requested(Request) with the meaning “agent identified by Request.AgentId is requesting relevant plan instances for Request.Desire”, or provided(AgentId, Request, Instances) with the meaning “agent identified by AgentId has cooperated in response of the request Request by providing a set Instances of relevant plan instances for Request.Desire”. In both cases Desire has the form achieve(Atom), where Atom may contain variables.

Ordinary events include at least the following ones:

- received(AgentId, Message) with the meaning “message Message has been received from the agent identified by AgentId”;
- added(Belief) with the meaning “the new belief Belief has been added to the agent’s data base”;
- removed(Belief) with the meaning “the belief Belief has been removed from the agent’s data base”.

Events perceived from the agent are stored in a priority queue.

Agents and cooperations. An agent is defined by the following data:

- a unique identifier TheAgentId;
- a priority queue TheEventQueue containing the events perceived by the agent;
- a set of desires TheDesires containing all main desires to be achieved;
- a set of specific plans TheSpecificPlans;
- a non empty set of default plans TheDefaultPlans;
- a set of intentions TheIntentions;
- a set of beliefs TheBeliefs;
- a cooperation TheCooperation specifying a set of trusted agent identifiers TrustedAgents, a plan retrieval policy Retrieval and a plan acquisition policy Acquisition. A plan retrieval policy ranges over the two values always and noLocal, whereas a plan acquisition ranges over the three values discard, add and replace;
- a relation DesireIntention between desires and intention identifiers which must be one-to-one between TheDesires and the set of intention identifiers IntentionIdentifiers = \{Id \ s.t. \ \exists \ \text{Intention} \in \text{TheIntentions} \land \text{Intention.IntentionId} = Id\};
– a relation `IntentionRequest` between intention identifiers and requests s.t. the identifier of any suspended intention in `TheIntentions` is associated with at most one request, and any request is associated with at most one intention identifier identifying a suspended intention in `TheIntentions`;
– a constant predicate `CanResume` taking a cooperation request, a set of agent identifiers, and a set of plan instances;
– a constant total function `GetDesires` taking an ordinary event and returning a possibly empty set of desires;
– a constant partial function `SelectInstance` taking a possibly empty set of plan instances and returning a member of such a set (the function is undefined if such a set is empty). The function must meet the following requirement: if `SelectInstance(Instances)` returns an instance of a default plan, then `Instances` does not contain instances of specific plans;
– a constant total function `SelectIntention` taking a non empty set of active intentions and returning a member of such a set;
– a constant total function `SelectState` taking a non empty set of states and returning a member of such a state.

### 3.3 Primitive Operations

The primitive operations used in the specification are listed below. For each data structure, `Empty` is the boolean primitive checking whether its argument is empty.

– Primitives on the event queue: `Empty`, `Get` and `Put`.
– Primitives on sets and relations: `\emptyset`, `\cup`, `\setminus`, `\in`.
– Primitives on intention stacks: `EmptyStack` (constant empty stack), `Empty`, `Push`, `Pop`.
– primitives on internal actions sequences: `Empty`, `Head`, `Tail`.
– Primitives for collaborative plan exchange and for sociality:
  - `RequestOp`: takes an agent identifier `AgentId` and a request `Request` and puts `requested(Request)` into the event queue of `AgentId`.
  - `ProvideOp`: takes an agent identifier `ProviderId`, a request `Request` and a set of instances `Instances` and puts `provided(ProviderId,Request,Instances)` into the event queue of `Request,AgentId`.
  - `SendOp`: takes two agent identifiers, `SenderId` and `ReceiverId`, and a message `Message`, and puts `received(SenderId,Message)` into the event queue of `ReceiverId`. Currently we do not commit to any agent communication language: `Message` may be any kind of logical expression eventually containing free variables implementing information passing.
– Logical primitives:
  - `Ground`: takes an atom as its argument and returns `true` if it does not contain variables, `false` otherwise.
  - `Apply`: takes a substitution `\theta` and a term `t` and returns `t\theta`, namely the term obtained from `t` by replacing each variable `x` by `\theta(x)`.
• \textit{Compose} takes two substitutions \( \sigma \) and \( \theta \) and returns the substitution \( \sigma \theta \) s.t. for every variable \( x \), \( (\sigma \theta)(x) = (x \sigma) \theta \).

• \textit{Mgu}: takes two expressions and returns their most general unifier.

• \( \models \): takes a couple \( (S, A) \) such that \( S \) is a set of atoms and \( A \) is an atom and returns true if \( A \) is a logical consequence of \( S \), false otherwise.

3.4 Behavioral Specification

The behavioral specification of Coo-BDI is defined using an informal functional language based on Extended ML [10] including boolean connectors and existential and universal quantifiers. Formalizing the given specification using standard notations is part of our future work.

Due to space constraints we will omit the specification of those auxiliary functions whose behavior can be expressed in natural language. We will also skip the details of processing active intentions since, w.r.t. dMARS, this step has been only marginally modified by the introduction of cooperations. The interested reader can find the complete specification of Coo-BDI in the technical report downloadable from ftp://ftp.disi.unige.it/pub/person/AnconaD/BCDI.ps.gz.

	extbf{Processing Events}. If the event queue is empty, the successive engine step is performed. Otherwise, an event \textit{Event} is taken from the event queue.

\begin{verbatim}
if not Empty(TheEventQueue) then
    let Event = Get(TheEventQueue)

Three situations may occur:
1) If \textit{Event} is of kind \textit{requested(Request)}, then \textit{GetRelInstances(Request)} retrieves all the instances of specific plans relevant for \textit{Request.Desire} and whose specifier is either public or only(\textit{TrustedAgents}) and \textit{TrustedAgents} includes \textit{Request.AgentId}. The retrieved \textit{Instances} set is posted into the \textit{Request.AgentId}'s event queue by calling \textit{ProvideOp(\textit{TheAgentId},Request,Instances)}.

    if \exists \textit{Request.s.t. Event = requested(Request)} then
        let Instances = GetRelInstances(Request)
        ProvideOp(\textit{TheAgentId},Request,Instances)

\forall \textit{Plan,Substitution,Request}
    (\textit{Plan,Substitution}) \in GetRelInstances(Request) \Rightarrow
    Plan \in \textit{TheSpecificPlans} \land
    (\text{Plan,AccessSpecifier = public} \lor
    (3\textit{AgentIds} s.t. \text{Plan,AccessSpecifier = only(\textit{AgentIds})} \land \text{Request.AgentId} \in \text{AgentIds}) \land
    \text{Substitution = Mgu(Plan,Trigger,Request,Desire)})

2) \textit{Event} is of kind \textit{provided(AgentId,Request,Instances)}. If there exists \textit{Intention} s.t. \( (\text{Intention,IntentionId,Request}) \in \text{IntentionRequest} \) (the retrieval of relevant plan instances for \textit{Request.Desire} is still ongoing) \textit{Intention} is updated by adding the retrieved \textit{Instances} to the \textit{RelevantInstances} field and by removing the identifier of the answering agent \( \text{(AgentId)} \) from the \text{WaitingOn} field. The updated intention replaces \textit{Intention} in \textit{TheIntentions}.
\end{verbatim}
else if \( \exists \) AgentId, Request, Instances s.t.
\( \text{Event} = \text{provided}(\text{AgentId}, \text{Request}, \text{Instances}) \) then
   if \( \exists \) Intention s.t. (Intention, IntentionId, Request) \( \in \) IntentionRequest then
      let UpdatedIntention = Intention
      UpdatedIntention. RelevantInstances :=
      Intention. RelevantInstances \( \cup \) Instances
      UpdatedIntention. WaitingOnAgents :=
      Intention. WaitingOnAgents \( \setminus \) \{AgentId\}
      TheIntentions := (TheIntentions \( \setminus \) Intention) \( \cup \) UpdatedIntention
   end if
end if

3) If Event is an ordinary event, the set of corresponding desires is evaluated by means of getDesires(Event) and the desire set is updated to contain them. For each generated desire, a new intention stack is created and initialized (CreateIntention), the intention set and the DesireIntention relation are updated to take into account the newly created intention and the retrieval of relevant plan instances for the intention and the desire is started (RetrieveRelevantInstances(Intention, Desire)).

else let NewDesires = getDesires(Event)
TheDesires := TheDesires \( \cup \) NewDesires
\( \forall \) Desire \( \in \) NewDesires
   \( \exists \) Intention s.t. CreateIntention(Intention)
   let UpdatedIntention = RetrieveRelevantInstances(Intention, Desire)
TheIntentions := TheIntentions \( \cup \) UpdatedIntention
DesireIntention := DesireIntention \( \cup \) (Desire, UpdatedIntention, IntentionId)

When a new intention is created, an unused identifier is assigned to it, its RelevantInstances, WaitingOnAgents and FailedInstances fields are set to the empty set, and its status is set to suspended.

Updating an intention by retrieving the relevant plan instances for the desire that the intention is currently achieving (either main or subaltern) means setting the intention status to suspended, getting the instances of local plans relevant for the desire (GetLocalRelevantInstances(Desire)), setting the RelevantInstances field to the returned set and updating the WaitingOnAgents fields according to the plan retrieval policy. If the policy is always or it is noLocal and no local relevant instances of specific plans are found (at least one instance of one default plan is always found by definition of default plan) then the WaitingOnAgents field is set to the trusted agents set, a plan request for the desire is issued to each trusted agent (RequestOp(AgentId, Request)) and the IntentionRequest relation is updated. Otherwise, WaitingOnAgents is set to the empty set.

\( \forall \) UpdatedIntention, Intention, Desire, Request
UpdatedIntention = RetrieveRelevantInstances(Intention, Desire) \( \iff \)
UpdatedIntention. IntentionId = Intention. IntentionId \^
UpdatedIntention. Stack = Intention. Stack \^
UpdatedIntention. FailedInstances = Intention. FailedInstances \^
UpdatedIntention. Status = suspended \^
UpdatedIntention. RelevantInstances = GetLocalRelevantInstances(Desire) \^
(((TheCooperation. Retrieval = always) \^ SpecificPlan(UpdatedIntention. RelevantInstances) = \emptyset) \^
UpdatedIntention. WaitingOnAgents = TheCooperation. TrustedAgents \^
Request = CreateRequest(AgentId, Desire) \∧
∀ AgentId ∈ TheCooperation.TrustedAgents
   RequestOp(AgentId, Request) \∧
IntentionRequest =
   IntentionRequest \(\langle\) UpdatedIntention, IntentionId, Request \(\rangle\) \∧
   (TheCooperation.Retrieve = noLocal \∧
    SpecificPlans(UpdatedIntention.RelevantInstances) ≠ ∅ \∧
    UpdatedIntention.WaitingOnAgents = ∅))

∀ Plan, PlanInstancesSet
   Plan ∈ SpecificPlans(PlanInstancesSet) \implies
     ∃ Substitution s.t. (Plan, Substitution) ∈ PlanInstancesSet \∧
     Plan = TheSpecificPlans

∀ Plan, Substitution, Instances
   (Plan, Substitution) ∈ GetLocalRelInstances(Desire) \implies
     Plan = TheSpecificPlans \∧
     Substitution = Mgu(Plan.Trigger, Desire) \∨
     Plan = TheDefaultPlans \∧
     Substitution = ∅

∀ Request
   Request = CreateRequest(AgentId, Desire) \iff
     Request. RequestId = 1 \∧
     Another ∈ ran(IntentionRequest) s.t. Another. RequestId = 1 \∧
     Request. AgentId = AgentId \∧
     Request. Desire = Desire

**Processing Suspended Intentions.** During this step the engine checks if there are suspended intentions which can be resumed since the associated Request and the WaitingOnAgents and RelevantInstances fields satisfy the CanResume condition.

∀ Intention ∈ TheIntentions s.t. Intention. Status = suspended
   if ∃ Request s.t.
     (Intention. IntentionId, Request) ∈ IntentionRequest \∧
     CanResume(Request, Intention. WaitingOnAgents, Intention. RelevantInstances)

If one resumable intention is found, two cases must be considered:
1) The intention stack is empty: the desire for which the relevant plan instances have been collected is a main desire. To implement backtracking on the plan instances which can be used to achieve the main desire, the already failed plan instances (Intention. FailedInstances) are not re-attempted. The applicable plan instances are thus evaluated starting from the collected relevant instances Intention. RelevantInstances minus the failed instances.
2) The intention stack is not empty: the desire for which the relevant plan instances have been collected is a subaltern desire, and no backtracking is implemented for it. The applicable plan instances are thus evaluated starting from Intention. RelevantInstances.

then
   if Empty(Intention. Stack) then
     let Applicable = GetAppInstances(
       Intention. RelevantInstances \Intention. FailedInstances)
   else
     let Applicable = GetAppInstances(Intention. RelevantInstances)
GetAppInstances(ReqInst) returns all the instances (Plan, Substitution) obtained from ReqInst such that the plan precondition instantiated with Substitution is a ground logical consequence of the agent’s beliefs. By definition, for any desire there is at least one relevant plan instance (the one originated by one default plan), and this instance is also applicable (there is no precondition to check): when GetAppInstances is applied to a non-empty set of relevant plan instances it always returns at least one plan instance; it returns the empty set if and only if its argument is the empty set. In the specification given so far, Applicable may be empty only if the intention relevant instances minus the intention failed instances is the empty set, namely, all the relevant instances for the main desire have failed. In this case, the desire fails: both the desire and the intention must be removed by the corresponding sets and the relations involving them must be updated.

if Applicable ≠ ∅ then
  TheDesires := TheDesires \ {Request, Desire}
  TheIntentions := TheIntentions \ {Intention}
  DesireIntention := DesireIntention \ 
  \{(Request, Desire, Intention, IntentionId)
  IntentionRequest := IntentionRequest \ \{(Intention, IntentionId, Request)}

If Applicable is not empty, a plan instance is selected from it and its execution is created by setting the instance field to the selected plan instance, the substitution field to Substitution, the current state to the root of the plan body and the next states to the children of the current state. The intention execution is pushed onto the intention stack and the intention’s status specifier is set to active. The selected plan is discarded, added to the plan library or used to replace all the existing plans whose trigger unifies with the plan’s trigger to which Substitution is applied according to the agent’s plan acquisition policy.

else
  let Instance = SelectInstance(Applicable)
  let Execution = StartExec(Instance)
  let UpdatedIntention = Intention
  UpdatedIntention.Stack := Push(Execution, Intention.Stack)
  UpdatedIntention.Status := active
  TheIntentions := (TheIntentions \ {Intention}) \ UpdatedIntention
  if Instance. Plan ≠ TheSpecificPlans then
    TheSpecificPlans := AcquirePlan(Instance)

∀ Plan, Subst, Instances
(Plan, ComposedGroundingSubst) ∈ GetAppInstances(Instances) ⇔
(Plan, Subst) ∈ Instances ∧
∃ GroundingSubst s.t.
GroundLogicalConsequence(GroundingSubst, Apply(Subst, Plan, Precondition)) ∧
GroundingSubst ≠ null ∧
ComposedGroundingSubst = Compose(Subst, GroundingSubst)

∀ Plan, Substitution, Instance, Execution
InstanceExecution = StartExec((Plan, Substitution)) ⇔
InstanceExecution. Instance = (Plan, Substitution) ∧
Processing Active Intentions. Active intentions are processed like in dMARS. If there are active intentions, one is selected and removed by the intention set (a modified copy will be added at the end of the management) and the topmost plan instance execution is taken. Three cases may arise. 1) The current state of the topmost plan instance execution is a leaf: the plan succeeds and the procedure for making the situation is called. 2) The invariant condition of the plan associated with the plan instance instantiated with the execution substitution is not satisfied in the current agent’s state or the current state of the topmost plan instance execution is not a leaf, and there are no more states reachable from it: the plan fails and the procedure for managing this situation is called. 3) The topmost plan instance execution neither fails nor succeeds: the action to perform, which labels the edge between the current state and the selected successive state, is retrieved. If the action is add(Atom), remove(Atom), send(AgentId,Message) or query(SituationFormula), the calls to primitive operations and updates to the intention, belief base and event queue are performed almost like in dMARS. If the action is achieve(Desire) and the desire is not a logical consequence of the agent’s current beliefs, RetrieveRelInstances is called.

4 Related Work and Conclusions

Starting from the BDI model, many extensions have been proposed in literature. Most of them add some attitude or ability to the basic BDI theory or provide a better formalization of the relationships between existing ones. Just to make some examples, [13] extends the theoretical BDI framework with the notion of capability and investigates how capabilities affect the agent’s reasoning about intentions. In [2] the BODI (Beliefs, Obligations, Intentions, Desires) architecture is discussed. It contains feedback loops to consider all effects of actions before committing to them, and mechanisms to resolve the conflicts between the outputs of the B, O, I, D components. In [5], classical BDI agents are extended with conditional mental attitudes represented by interconnected components. Mental attitudes are considered to be functions of other mental attitudes and functional dependencies between them are analyzed. The paper [15] investigates the impact that sociality has on mental states and how it can be formalized within a BDI
model, while [14] focuses on actions and their formalization by adding three more operators to the basic BDI ones: capabilities, opportunities and results. All these works are more concerned with the logical formalization of the theory behind the extended BDI model than with the complete specification of an implementable system, as our work is.

More similar in spirit to our pragmatic approach are the attempts to extend BDI systems with mobility. In [3] the TOMAS (Transaction Oriented Multi Agent System) architecture is described: it combines the distributed nested transaction paradigm with the BDI model. An algorithm is presented which enables agents in TOMAS to become mobile. The JAM BDI-theoretic mobile agent architecture [8] provides plan and procedural representations, metalevel and utility-based reasoning over simultaneous goals and goal-driven and event-driven behavior. Mobility is realized by an agentGo primitive for agent migration that may appear in the body of plans.

To the best of our knowledge, however, there are no extensions to the BDI model or to BDI-based implemented systems that allow agents to cooperate by exchanging plans: in all the BDI-based systems, theories and languages we are aware of, the plan library is a static component and there are no means to extend the agent’s procedural knowledge at runtime. Our proposal overcomes this limitation by extending the architecture interpreter. This means that the agent’s developer may take advantage of the cooperativity of agents for free: all the hard work of providing and retrieving external plans is managed by the interpreter without the agent’s developer needing to be aware of the details of this mechanism. A classic BDI agent can be modeled in Coo-BDI by setting the access specifier of all its plans to private and by setting the trusted agents set to the empty set. In this way, the agent will never look for external relevant plans, and will never share its own plans, exactly as it happens in the original BDI model.

As far as the future work on Coo-BDI is concerned, there are three main directions we will follow: 1) providing a better formalization on Coo-BDI using a standard specification language; 2) analyzing how exchanges of plans can be described as part of a more general speech-act based communication mechanism adopting standard languages such as FIPA ACL (http://www.fipa.org/) and KQML [11]; and 3) developing a working Coo-BDI interpreter and using it to implement one of the applications discussed in the introduction. This exercise will allow us to get feedbacks on the Coo-BDI formalization, (possibly) confirming the adequacy of the approach presented in this paper.

Acknowledgments. The authors thank Rafael Bordini, Mehdi Dastani, Mark D’Inverno and Michael Winikoff for the useful discussions on the “no applicable plans” problem and the anonymous referees for their thoughtful and constructive comments.
References


