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 define Coo-BDI, an extension of 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 agents which do not possess their own procedural knowledge for processing a given event.

1 Introduction

Intelligent agents [27, 12] 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 [22], whose wide appreciation is witnessed by the development of a BDI logic [23], the definition of BDI-based languages (AgentTalk [26], 3APL [9], AgentSpeak(L) [21]) and the creation of BDI-based development tools such as JACK [6], PRS [17] and dMARS [10].

Despite this large consensus, some BDI concepts and mechanisms are not represented and implemented 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 occurs when events (or subgoals, since they are considered the same entity in implemented systems) cannot be managed because of the lack of suitable plans. 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 ignored; in others the definition of a default "catch-all" plan is required. This heterogeneity is partly due to the little attention received by the notion of *agent cooperativity* which, instead, should deserve a major role in the BDI architecture.

Cooperativity is the ability of an agent to help other agents to achieve their desires. While this feature needs to be implemented in a rather ad hoc way each time it must be exhibited by a BDI agent, Coo-BDI overcomes this problem by providing a rather powerful mechanism for promoting agent cooperativity, by allowing agents to exchange their plans. 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 loading and linking of code is a fundamental aspect. If the traditional BDI approach is adopted, implementing this feature 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. Coo-BDI 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, it replaces them with new code, provided by a server agent, without needing to stop its own activity or 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, Coo-BDI is suitable for modeling all kinds of "learning agents" which learn from the interaction with other agents. This is made possible by the main novel feature of Coo-BDI consisting of a built-in mechanism for exchanging plans between agents when there are no local applicable plans for achieving a certain desire[1]. As a consequence, the notions of *desire* and *event* must be kept clearly distinct; an agent has an event queue containing events generated by the environment or by other agents. Managing an event means removing it from the queue and generating the set of desires associated with it; note that such a process is inherently local, in the sense that each agent decides which are its own desires associated with a certain event without any interference from the outside (that is, there is no cooperativity at this stage).

[1] We prefer to use the term "desire" rather than "goal" for coherence with the Belief, Desires, Intentions acronym.

For example, the event "*reception of an e-mail asking to join the party at Liza's place at 7 p.m.*" may generate the desires "*cancel all the appointments for this evening*" and "*reach Liza's place at 7 p.m.*" in a given agent, and the desire "*phone Liza for thanking and declining*" in another agent. Therefore, desires must be kept distinct from events and maintained as explicit information in an ad hoc structure.

In Coo-BDI cooperativity can be exploited only after desires generation, when the appropriate plans for achieving the generated desires must be selected. Let us go on with the previous example, by supposing that the agent which desires to reach Liza's place at 7 p.m. has no procedural knowledge (namely, no plans) for achieving that 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 Liza's home. Some of the agents contacted for advice could cooperate by suggesting plans for achieving this desire, allowing the requiring agent to join the party.

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. In Section 4 we provide the structural specification of Coo-BDI and in Section 5 we provide its behavioral specification. The usefulness of our extension is shown by means of an example in Section 6. Section 7 outlines related and future work.

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 can not change neither can be exchanged 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 plan instances whose trigger event matches an event in the event queue (*relevant* plan instances) and whose precondition is satisfied (*applicable* plan instances);

3. select for execution one instance from the set of applicable plan instances;

4. push the selected instance 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 instance and execute the next step of this current instance: if the step is an action, perform it, otherwise, if it is a subgoal, insert 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 [22], Rao and Georgeff write that after the plan instance generator reads the event queue and returns a list of applicable instances, the deliberator selects from this list a subset of instances which are added to the intention structure. We can guess that if no instances can be adopted the interpreter simply jumps to the next step but this is not explicitly established by the specification. In [10], d'Inverno, Kinny, Luck and Wooldridge provide a more detailed specification, however, the case "no applicable plans found" is not considered. In the original specification of AgentSpeak(L) [21], it is not clear what the interpreter does if the set of applicable plan instances is empty. According to the more operational-oriented specification of AgentSpeak(L) given in [2, 15], whose brief description can be found in [16], an event should be discarded if it has no relevant plan instances.

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 instances, they simply ignore the event and remove it from the queue. Such an "ignore and discard" behavior can be modified in JACK and dMARS only by exploiting the ad hoc meta-programming features offered by the languages. The implementation of 3APL (`http://www.cs.uu.nl/3apl/`) adopts a default instance if no other instances can be applied to deal with the selected event. In the implementation of AgentSpeak(L) developed by Bordini et al. [1] (`http://www.inf.ufrgs.bf/~massoc`) the user has two possible options if there are no applicable instances for a given event: she/he can ask the interpreter either to discard the event or to keep it in the set of events. In the latter case, the event can be eventually managed if at a certain time it has an applicable plan instance.

This short overview shows that 1) BDI does not provide a built-in mechanism allowing plan exchange between agents and 2) in BDI systems the "no applicable instances" problem is either neglected or solved in rather naive ways. On the contrary, Coo-BDI promotes cooperation by allowing agents to exchange their plans and exploits such a feature for partly solving the "no applicable instances" problem in a rather natural and general way.

3 Coo-BDI: a gentle introduction

Coo-BDI (Cooperative BDI) is based on the dMARS specification [10] 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 *cooperations* among agents to retrieve external plans for achieving desires (both main and subaltern), the extension of plans with *access specifiers*, the introduction of *default* plans, the extension of *intentions* to take into account the external plan instances retrieval mechanism and the modification of the Coo-BDI *engine* (interpreter) 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 plans are classified in *specific* and *default* ones; like BDI plans, both kinds of plans consist of a trigger, a precondition, a body, an invariant and 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). A default plan is a plan where the access specifier is *private*, the trigger is a variable and the precondition is the *true* constant; therefore, by definition, a default plan can always be applicable and can never be exchanged. Each agent must always provide at least one default plan, so that every desire can be managed. A specific plan is simply a non default plan.

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, which is essential for understanding how steps 1 and 3 work. Such a mechanism consists of four sequential steps. a) The intention is suspended. b) The local relevant plan instances for the desire are generated and associated with the intention. c) According to the cooperation, the set S of the agents expected to cooperate is defined. d) A plan request for the desire is created and sent to all the agents in S.

Events in the 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 message reception and notification of updates performed on the agent's beliefs set. When an agent receives a request for a desire from another agent A, it sends to A the (possibly empty) set of all its local plan instances which are both relevant for that desire and visible to A (recall that, by definition, default plans cannot be visible). 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 instance is selected and the corresponding plan instance execution 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, active intentions are managed like in the BDI architecture except for the mechanism for retrieving external relevant plans triggered by desire achievement.

The reader may find the structural specification of Coo-BDI in Section 4 and its behavioral specification in Section 5. Both Coo-BDI data structures and behavior are specified using Prolog.

4 Structural Specification of Coo-BDI

The structural specification of Coo-BDI is given by means of a BNF. We will use the following informally defined non-terminals:

- `Variable`: a variable symbol
- `Pred`: a predicate symbol
- `Term`: a term
- `GroundTerm`: a ground term
- `AgentId`: a string representing the identity of an agent
- `RequestId`: a string identifying a request
- `IntentionId`: a string identifying an intention
- `Message`: any kind of term eventually containing free variables
- `Substitution`: a mapping from variables to terms
- Every non terminal NT followed by Set (eg: `AgentIdSet`) represents a set of elements in NT.
- Every non terminal NT followed by Sequence (eg: `InternalActionSequence`) represents a sequence of elements in NT.
- Every non terminal NT followed by Queue (eg: `EventQueue`) represents a queue of elements in NT.
- Every non terminal NT followed by Stack (eg: `PlanInstanceExecutionStack`) represents a stack of elements in NT.

Clearly, the only difference for the last four items consists in the primitives that will be used.

Beliefs, desires, queries and actions. A belief formula is either an atom or the negation of an atom. Beliefs are ground belief formulas. Desires have the form `achieve(BeliefFormula)`. Queries are denoted by `query(SituationFormula)` where a situation formula is either a belief formula, 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 data base of an agent's beliefs: `add(BeliefFormula)` and `remove(BeliefFormula)`. These actions can only be executed if their argument is ground. External actions include at least the ability of sending messages to agents: `send(AgentId, Message)`.

```
BeliefFormula ::= Pred(Term, ...., Term) | not(Pred(Term, ...., Term))
Belief ::= Pred(GroundTerm, ...., GroundTerm) |
  not(Pred(GroundTerm, ...., GroundTerm))
SituationFormula ::= true | false | BeliefFormula |
  SituationFormula and SituationFormula |
  SituationFormula or SituationFormula
Desire ::= achieve(BeliefFormula)
Query  ::= query(SituationFormula)
InternalAction ::= add(BeliefFormula) | remove(BeliefFormula)
ExternalAction ::= send(AgentId, Message)
Action ::= InternalAction | ExternalAction
```

Plans. There are no syntactic differences between specific and default plans: both are defined by an access specifier, a trigger, a precondition, a body, an invariant, and two sets of internal actions to be used respectively when the plan succeeds or fails.

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. 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. Success and failure actions are sequences of internal actions.

```
AccessSpecifier ::= private | public | only(AgentIdSet)
EdgeLabel ::= Desire | Query | Action
Body ::= state(EdgeBodySequence)
EdgeBody ::= (EdgeLabel, Body)
Plan ::= plan(AccessSpecifier, Desire, SituationFormula, Body,
SituationFormula, InternalActionSequence, InternalActionSequence)
```

Plan instances. A plan instance is a pair `(Plan, Substitution)` formed by a plan and a substitution.

A plan instance `(Plan, Substitution)` is called *relevant* w.r.t. a desire if `Substitution` is a most general unifier for that desire and the trigger of `Plan`. Note that plan instances formed by a default plan are always relevant.

A plan instance `(Plan, Substitution)` is called *applicable* if the formula obtained by applying `Substitution` to the precondition of `Plan` is a logical consequence of the beliefs of the agent's data base. Note that plan instances formed by a default plan are always applicable.

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

```
PlanInstance ::= (Plan, Substitution)
PlanInstanceExecution ::=
planInstanceExecution(PlanInstance, Substitution, State, StateSet).
```

Intentions and requests. An intention is composed by a unique identifier, a stack of executions of plan instances, a status, a set of relevant plan instances, a set of agent identifiers and a set of failed plan instances.

The intention identifier is used for modeling in a convenient way the two relations `DesireIntention` and `IntentionRequest`, represented by the two predicates `relationDesireIntention(Desire, IntentionId)` and `relationIntentionRequest(IntentionId, Request)`, which associate with each main desire exactly one corresponding intention and with each suspended intention the corresponding (if any) request of relevant plan instances, respectively.

The stack of executions of plan instances is similar in spirit to the execution stack of logic programs.

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 for which no plan instance has been selected yet; in this case the relevant instances set contains the relevant plan instances which have been already collected for the desire, and the set of agent identifiers contains all the identifiers of those agents which are still expected to cooperate for achieving the desire.

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

A relation `relationIntentionRequest` associates with any intention identity of suspended intentions its current (if any) request for cooperation, and with any request the suspended intention it originates from, if the request is still valid.

```
Intention ::= intention(IntentionId, PlanInstanceExecutionStack, Status,
  PlanInstanceSet, AgentIdSet, PlanInstanceSet)
Status ::= suspended | active
Request ::= request(RequestId, AgentId, Desire)
```

Events. There are two kinds of events: cooperation and ordinary events. A cooperation event is either `requested(request(RequestId, ReqAgentId, Desire))` with the meaning "agent identified by `ReqAgentId` is requesting relevant plan instances for `Desire`", or `provided(AgentId, request(RequestId, ReqAgentId, Desire), Instances)` with the meaning "agent identified by `AgentId` has cooperated in response of the request `request(RequestId, ReqAgentId, Desire)` by providing a set `Instances` of relevant plan instances for `Desire`".

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 `eventQueue(EventQueue)`.

```
Event ::= CooperationEvent | OrdinaryEvent
CooperationEvent ::= requested(request(RequestId, AgentId, Desire)) |
 provided(AgentId, request(RequestId, AgentId, Desire), PlanInstanceSet)
OrdinaryEvent ::= received(AgentId, Message) | added(Belief) |
 removed(Belief)
```

Agent definition. An agent is defined by the following predicates (defined in the agent's knowledge base):

- `agentId(AgentId)` specifying the unique agent's identifier;
- `eventQueue(EventQueue)` specifying the current agent's event queue;
- `isDesire(Desire)` specifying the current set of agent's main desires;
- `isPlan(Plan)` specifying the current set of (both specific and default) agent's plans;
- `isIntention(Intention)` specifying the current set of agent's intentions;
- the current agent's beliefs;
- three predicates specifying the current agent's *cooperation*:
  - `trustedAgents(TrustedAgents)` specifying the current set of identifiers of the agents trusted by the agent;
  - `retrievalPolicy(Retrieval)` specifying the current retrieval policy, where `Retrieval ::= always | noLocal`;
  - `acquisitionPolicy(Acquisition)` specifying the current plan acquisition policy, where `Acquisition ::= discard | add | replace`.
- `relationDesireIntention(Desire,IntentionId)` specifying a one-to-one relation between all the current main desires of the agent and the identifiers of all its current intentions;
- `relationIntentionRequest(IntentionId,Request)` specifying a one-to-one relation between the identifiers of some of the currently suspended intentions of the agent and some requests issued by the agent;
- `canResume(Request, AgentIdentifiers, PlanInstances)` specifying whether a certain suspended intention, waiting on the cooperation request `Request`, can be resumed, provided that the set of agents which still have to reply to the request is `AgentIdentifiers`, and the set of relevant plan instances collected so far is `PlanInstances`;
- `getDesires(OrdinaryEvent, Desires)` specifying a total function mapping every ordinary event into the (possibly empty) set of main desires that must be generated from that event;
- `selectInstance(PlanInstances, SelectedPlanInstance)` specifying a total function returning a specific member `SelectedPlanInstance` of each non empty sets of plan instances `PlanInstances`, s.t. if `SelectedPlanInstance` is an instance of a default plan, then `PlanInstances` does not contain instances of specific plans;
- `selectIntention(Intentions, SelectedIntention)` specifying a total function returning a specific member `SelectedIntention` of each non empty set of active intentions;
- `selectState(States, SelectedState)` specifying a total function returning a member `SelectedState` of each non empty set of states `States`.

Among the predicates listed above, `agentId`, `canResume`, `getDesires`, `selectInstance`, `selectIntention` and `selectState` are *static* in the sense that they cannot be modified during the agent's lifetime, whereas all the others can be dynamically modified.

5 Behavioral Specification of Coo-BDI

The behavioral specification of Coo-BDI is given using Prolog.

Besides obtaining a working prototype for free, using a logic programming language for specifying a multi-agent system has other advantages:

- *MAS execution*: the evolution of a MAS consists of a nondeterministic succession of events; from an abstract point of view a logic programming language is a nondeterministic language in which computation occurs via a search process.
- *Meta-reasoning capabilities*: agents may need to dynamically modify their behavior so as to adapt it to changes in the environment. Thus, the possibility given by logic programming of viewing programs as data is very useful in this setting.
- *Rationality and reactiveness of agents*: the *declarative* and the *operational* interpretation of logic programs are strictly related to the main characteristics of agents, i.e., *rationality* and *reactiveness*. In fact, we can think of a *pure* logic program as the specification of the rational component of an agent and we can use the operational view of logic programs (e.g. left-to-right execution, use of non-logical predicates) to model the reactive behavior of an agent. The adoption of logic programming for combining reactivity and rationality is described in [13].

These advantages motivate our choice to use Prolog to specify Coo-BDI. Finally, note that the set of clauses specified in the sequel should be considered as part of a formal (executable) specification, rather than a Prolog program implementing an interpreter for Coo-BDI agents. Therefore, we have preferred to sacrifice implementation issues and efficiency in favor of a better clarity.

Notations and Assumptions. Variables begin with an uppercase letter or with _ (we use the last notation for unnamed variables which appear only once in a clause).

We exploit meta-logical, extra-logical and second-order predicates of the language, as well as red cuts to implement negation.

- *Meta-logical predicates.* We use `var(Term)` to test whether `Term` is a variable or not; `Term1\=Term2` to test not unificability of terms `Term1` and `Term2`; `Term1==Term2` to unify terms `Term1` and `Term2`; `call(Goal)` to call `Goal`; `(Goal1; Goal2)` to denote the logical disjunction of `Goal1` and `Goal2`.
- *Extra-logical predicates.* To specify updates to the agent state, we use the standard predicates `assert(Fact)`, `retract(Fact)` and the non standard `retractall(Fact)` which always succeeds and retracts all clauses of a dynamic predicate whose head unifies with `Fact`. The use of update primitives can introduce incoherence in the knowledge base if they appear within a failing goal. To avoid this undesirable effect and keep the semantics of our specification clear, we always leave update primitives at the end of the goal; furthermore, `retract(Fact)` is always called after checking that `Fact` is present in the knowledge base, in order to avoid undesired failures.
- *Second-order predicates.* To specify the retrieval of a set of elements satisfying a given condition (goal), we use the `setof` standard predicate, s.t.

setof(Template, Goal, Set) succeeds if Set is the set of all instances of Template such that Goal is satisfied, where that set is non empty. For our convenience, we assume that an empty set constant emptySet is returned by setof when no instances satisfying Goal are found.
- *Negation.* We use a not(Fact) predicate to express negation. Its implementation can be given using cut and fail.

We assume that a set of standard primitives on commonly used data structures is defined. We avoid discussing their meaning when it is clear or well-known.

- *Primitives on queues.* emptyQueue (constant empty queue), get(EventQueue, Element, ResultingEventQueue), put(Element, EventQueue, ResultingEventQueue).
- *Primitives on sets and relations.* emptySet (constant empty set), setUnion(Set1, Set2, Set1UnionSet2), singleton(Element, SetContainingOnlyElement), setDifference(Set1, Set2, Set1MinusSet2), belongs(Element, Set).
- *Primitives on stacks.* emptyStack (constant empty stack), push(Element, Stack, ResultingStack), pop(Stack, ResultingStack), top(Stack, TopElement).
- *Primitives on trees.* root(Tree, Node), children(Tree, Node, Set), leaf(Tree, Node), label(Tree, NodeOrigin, NodeDestination, LabelOfArcFromOriginToDestination).
- *Primitives on sequences.* emptySeq (constant empty sequence), head(Sequence, Head, RemainingSequence), tail(Sequence, RemainingSequence).

We assume that a set of Coo-BDI-specific primitives for plan exchange and for sociality is provided:

- multicastRequestOp(AgentIdSet, Request): takes a set of agent identifiers AgentIdSet and a request Request and puts requested(Request) into the event queue of each agent in AgentIdSet;
- provideOp(ProviderId, request(RequestId, AgentId, Desire), Instances): takes an agent identifier ProviderId, a request request(RequestId, AgentId, Desire) and a set of instances Instances and puts provided(ProviderId, request(RequestId, AgentId, Desire), Instances) into the event queue of AgentId;
- sendOp(SenderId, ReceiverId, Message): 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 conform to any specific agent communication language: Message may be any kind of term eventually containing free variables implementing information passing.

Finally, we assume that a primitive newId(Identity) for generating new fresh identities is available as well as a set of logical primitives. We refer to [24] for the notion of substitution, composition of substitutions and most general unifier.

- ground(Literal): true if Literal (Atom or not(Atom)) is ground.
- apply(Theta, Term, TermTheta): takes a substitution Theta and a term Term and returns the term obtained from Term by replacing each variable X by Theta(X).

– `compose(Sigma, Theta, SigmaTheta)`: takes two substitutions `Sigma` and `Theta` and returns their composition.
– `mgu(Expr1, Expr2, Mgu)`: takes two expressions and returns their most general unifier.

Coo-BDI engine. As already said in Section 3, the engine of Coo-BDI departs from the classical one to take into account both desires generation and cooperations, and is characterized by processing the event queue, processing suspended intentions and processing active intentions.

```
cooBDIengine :- processEventQueue, processSuspendedIntentions,
  processActiveIntentions.
```

Processing Events. If the event queue is empty, nothing is done (first clause). Otherwise, an event `Event` is taken from the event queue. The event is managed and the agent's event queue is updated.

```
processEventQueue :- eventQueue(EventQueue), empty(EventQueue).

processEventQueue :- eventQueue(EventQueue), not(empty(EventQueue)),
  get(Event, EventQueue, RemainingEventQueue), manageEvent(Event),
  retract(eventQueue(EventQueue)),
  assert(eventQueue(RemainingEventQueue)).
```

When managing an event, three situations may occur: 1) If `Event` is of kind `requested(request(RequestId, RequestingAgentId, Desire))`, then `getRelInstances(Request)` retrieves all the instances of specific plans relevant for `Desire` and whose specifier is either public or `only(TrustedAgents)`, and `TrustedAgents` includes `RequestingAgentId`. The retrieved `Instances` set is posted into the `RequestingAgentId` event queue by calling `ProvideOp(AgentId, Request, Instances)`, where `AgentId` is the identifier of the agent performing the `provideOp` action.

```
manageEvent(requested(request(ReqId, ReqAgentId, Desire))) :-
  agentId(AgentId),
  getRelInstances(Instances, request(ReqId, ReqAgentId, Desire)),
  provideOp(AgentId, request(ReqId, ReqAgentId, Desire), Instances).

getRelInstances(Instances, request(_ReqId, ReqAgentId, Desire)) :-
  setof((plan(AccessSpecifier, Trigger, Precondition, Body, Invariant,
              SuccessActions, FailureActions), Substitution),
        (isPlan(plan(AccessSpecifier, Trigger, Precondition, Body,
                     Invariant, SuccessActions, FailureActions)),
        (AccessSpecifier == public;
          (AccessSpecifier == only(TrustedAgents),
             belongs(ReqAgentId, TrustedAgents))),
           mgu(Trigger, Desire, Substitution)), Instances).
```

2) Event is of kind `provided(ProvidingAgentId, request(RequestId, AgentId, Desire), RetrievedInstances)`. If there exists an intention `Intention` s.t. the intention identifier and the request `request(RequestId, AgentId, Desire)` belong to the `relationIntentionRequest` then the retrieval of relevant plan instances for

`Desire` is still ongoing. `Intention` is updated by adding the retrieved `Instances` to the relevant instances and by removing the identifier of the answering agent (`ProvidingAgentId`) from the set `WaitingOnAgents`. The updated intention replaces `Intention` (first clause). Otherwise, if there is no `Intention` s.t. the intention identifier and the request belong to the `relationIntentionRequest`, the event is ignored (not shown).

```
manageEvent(provided(ProvidingAgentId, request(ReqId, AgentId,
        Desire), Instances)) :-
 relationIntentionRequest(IntentionId, request(ReqId, AgentId, Desire),
 isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
   WaitingOnAgents, FailedInstances)),
 singleton(ProvidingAgentId, SingletonProvidingAgentId),
 setDifference(WaitingOnAgents, SingletonProvidingAgentId,
   UpdatedWaitingOnAgents),
 setUnion(Instances, RelevantInstances, UpdatedRelevantInstances),
 retract(isIntention(intention(IntentionId, Stack, Status,
   RelevantInstances, WaitingOnAgents, FailedInstances))),
 assert(isIntention(intention(IntentionId, Stack, Status,
   UpdatedRelevantInstances, UpdatedWaitingOnAgents, FailedInstances))).
```

3) If `Event` is an ordinary event, the set of corresponding desires is retrieved by means of `getDesires(OrdinaryEvent, Desires)` and the main desire set is updated to contain them. For each generated desire not already present in the main desire set, 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)`).

```
manageEvent(Event) :- Event \= provided(_,_,_), Event \= requested(_),
  getDesires(Event, DesireSet), createIntentionsForDesires(DesireSet).

createIntentionsForDesires(DesireSet) :- belongs(Desire, DesireSet),
  not(isDesire(Desire)), assert(isDesire(Desire)),
  setDifference(DesireSet, Desire, RemainingDesireSet),
  createOneIntentionForOneDesire(Desire),
  createIntentionsForDesires(RemainingDesireSet).

createIntentionsForDesires(DesireSet) :- belongs(Desire, DesireSet),
 isDesire(Desire), setDifference(DesireSet, Desire, RemainingDesireSet),
 createIntentionsForDesires(RemainingDesireSet).
createIntentionsForDesires(emptySet).

createOneIntentionForOneDesire(Desire) :- newId(NewIntentionId),
  createIntention(NewIntentionId),
  retrieveRelevantInstances(NewIntentionId, Desire).
```

When a new intention is created its `RelevantInstances`, `WaitingOnAgents` and `FailedInstances` components are set to the empty set and its status is set to suspended.

```
createIntention(IntentionId) :-
  assert(isIntention(intention(IntentionId, emptyStack, suspended,
  emptySet, emptySet, emptySet))).
```

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, LocalRelevantInstances)`), setting the `RelevantInstances` component to the returned set and updating the `WaitingOnAgents` component according to the plan retrieval policy. If the retrieval policy is `always` or no local relevant instances of specific plans are found, the `WaitingOnAgents` component is set to the trusted agents set, a plan request for the desire is issued to each trusted agent (`multicastRequestOp(TrustedAgents, Request)`) and the `relationIntentionRequest` relation is updated (first clause). Otherwise, `WaitingOnAgents` is set to the empty set and no plan requests are sent (second clause).

```
retrieveRelevantInstances(IntentionId, Desire) :- agentId(AgentId),
  getLocalRelevantInstances(Desire, LocalRelInst),
  (retrievalPolicy(always); specificPlans(LocalRelInst, emptySet)),
  trustedAgents(TrustedAgents), newId(RequestId),
  multicastRequestOp(AgentId, TrustedAgents,
    request(RequestId, AgentId, Desire)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
    LocalRelInst, TrustedAgents, FailedInstances))),
  assert(relationIntentionRequest(IntentionId,
    request(RequestId, AgentId, Desire))).

retrieveRelevantInstances(IntentionId, Desire) :-
  getLocalRelevantInstances(Desire, LocalRelevantInstances),
  retrievalPolicy(noLocal),
  not(specificPlans(LocalRelevantInstances, emptySet)),
  retract(isIntention(intention(IntentionId, Stack, _Status,
    _RelevantInstances, _WaitingOnAgents, FailedInstances))),
  assert(isIntention(intention(IntentionId, Stack, suspended,
     LocalRelevantInstances, emptySet, FailedInstances))).

getLocalRelevantInstances(Desire, LocalRelInst):-
  setof((plan(AccessSpec, Trigger, Precondition, Body,
    Inv, SuccActions, FailActions), Subst),
    (isPlan(plan(AccessSpec, Trigger, Precondition, Body,
    Inv, SuccActions, FailActions)), mgu(Trigger, Desire, Subst)),
    LocalRelInst).

specificPlans(PlanInstancesSet, Plans) :-
  setof(plan(AccessSpec, Trigger, Precondition, Body,
    Inv, SuccActions, FailActions),
    (belongs((plan(AccessSpec, Trigger, Precondition, Body, Inv,
    SuccActions, FailActions), _Substitution), PlanInstancesSet),
```

```
    specific(plan(AccessSpec, Trigger, Precondition, Body, Inv,
    SuccActions, FailActions))), Plans).

specific(plan(AccessSpecifier, Trigger, Precondition, _Body,
  _Invariant, _SuccessActions, _FailureActions)) :-
  not((Precondition == true, var(Trigger), AccessSpecifier == private)).
```

Processing Suspended Intentions. During this step the engine checks if there are suspended intentions which can be resumed since the associated `Request`, the `WaitingOnAgents` component and the `RelevantInstances` component satisfy the `canResume` condition (first clause). If no resumable intention is found, `processSuspendedIntentions` succeeds without doing nothing (not shown).

```
processSuspendedIntentions :-
  isIntention(intention(IntentionId, _Stack, _Status, RelevantInstances,
  WaitingOnAgents, _FailedInstances)),
  relationIntentionRequest(IntentionId, Request),
  canResume(Request, WaitingOnAgents, RelevantInstances),
  resume(IntentionId, Request).
```

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 (the `FailedInst` component of the intention) are not re-attempted. The applicable plan instances are thus evaluated starting from the collected relevant instances `RelevantInst` minus the failed instances.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, emptyStack, _Status,
  RelevantInst, _WaitingOnAgents, FailedInst)),
  setDifference(RelevantInst, FailedInst, NotAttemptedInst),
  getApplInstances(ApplicableInstSet, NotAttemptedInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

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 `RelevantInst`.

```
resume(IntentionId, Request) :-
  isIntention(intention(IntentionId, Stack, _Status, RelevantInst,
  _WaitingOnAgents, _FailedInst)), not(empty(Stack)),
  getApplInstances(ApplicableInstSet, RelevantInst),
  manageApplicableInstances(ApplicableInstSet, IntentionId, Request).
```

The atom `getApplInstances(ApplicableInstSet, RelevantInst)` unifies `ApplicableInstancesSet` with the set of couples `(Plan, Substitution)` obtained from `RelevantInst` 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 (the precondition is always true): when `getApplInstances` 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, `ApplicableInstancesSet` 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.

```
getApplInstances(ApplicableInstancesSet, RelevantInstances) :-
  setof((plan(AccessSpecifier, Trigger, Precondition, Body, Invariant,
  SuccessActions, FailureActions), Substitution),
  (belongs((plan(AccessSpecifier, Trigger, Precondition, Body,
  Invariant, SuccessActions, FailureActions), Substitution),
  RelevantInstances),
  apply(Substitution, Precondition, InstantiatedPrecondition),
  call(InstantiatedPrecondition)), ApplicableInstancesSet).
```

In the applicable instances set is empty the desire fails: both the desire and the intention must be removed by the corresponding sets and the relations involving them must be updated.

```
manageApplicableInstances(emptySet, IntId, request(ReqId, AgId, Des)) :-
  isDesire(Des), isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances)),
  relationDesireIntention(Des, IntId),
  relationIntentionRequest(IntId, request(ReqId, AgId, Des)),
  retract(isDesire(Des)),
  retract(isIntention(intention(IntId, _Stack, _Status,
  _RelevantInstances, _WaitingOnAgents, _FailedInstances))),
  retract(relationDesireIntention(Des, IntId)),
  retract(relationIntentionRequest(IntId, request(ReqId, AgId, Des))).
```

If the applicable instances set 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. If the plan was already present in the plan library (first clause of acquire plan), nothing is done. Otherwise, according to the agent's plan acquisition policy, the selected plan is discarded (second clause), added to the plan library (third clause) or used to replace all the existing plans whose trigger unifies with the plan's trigger to which `Substitution` is applied (fourth clause).

```
manageApplicableInstances(ApplicableInstancesSet, IntentionId,
  _Request) :- ApplicableInstancesSet \= emptySet,
  selectInstance(ApplicableInstancesSet, (Plan, Substitution)),
  createExecution((Plan, Substitution), Execution),
  isIntention(intention(IntentionId, Stack, Status, RelevantInstances,
  WaitingOnAgents, FailedInstances)),
  push(Execution, Stack, UpdatedStack), acquirePlan(Plan),
  retract(isIntention(intention(IntentionId, Stack, Status,
  RelevantInstances, WaitingOnAgents, FailedInstances))),
```

```
  assert(isIntention(intention(IntentionId, UpdatedStack, active,
  RelevantInstances, WaitingOnAgents, FailedInstances))).

createExecution((plan(AccessSpecifier, Trigger, Precondition, Body,
  Invariant, SuccActions, FailActions), Substitution), Execution) :-
  root(Body, CurrentState), children(Body, CurrentState, NextStates),
  Execution == planInstanceExecution(
    (plan(AccessSpecifier, Trigger, Precondition, Body,
     Invariant, SuccActions, FailActions), Substitution),
     Substitution, CurrentState, NextStates).

acquirePlan(Plan) :- isPlan(Plan).

acquirePlan(Plan) :- not(isPlan(Plan)), acquisitionPolicy(discard).

acquirePlan(Plan) :- not(isPlan(Plan)), acquisitionPolicy(add),
  assert(isPlan(Plan)).

acquirePlan(plan(AccessSpecifier, Trigger, Precondition, Body,
  Invariant, SuccessActions, FailureActions)) :-
  not(isPlan(plan(AccessSpecifier, Trigger, Precondition, Body,
  Invariant, SuccessActions, FailureActions))),
  acquisitionPolicy(replace),
  retractall(isPlan(plan(_AccessSpecifier, Trigger, _Precondition,
  _Body, _Invariant, _SuccessActions, _FailureActions))),
  assert(plan(AccessSpecifier, Trigger, Precondition, Body, Invariant,
  SuccessActions, FailureActions)).
```

Processing Active Intentions. Active intentions are processed like in dMARS. For space constraints, we just provide an informal description, however the interested reader may find the complete formal specification at `ftp://ftp.disi.unige.it/pub/person/AnconaD/Coo-BDIspec.pl`.

If there are no active intentions, there is nothing to do, otherwise one is selected 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 instance succeeds and its execution is removed from the intention stack. The substitution associated with the execution is applied to the success internal actions sequence which can be executed thereafter. If there are no more elements in the stack, then the intention has succeeded and can be removed, as well as the main desire which generated it (the relation `relationDesireIntention` needs to be updated). Otherwise, the substitution of the removed execution is composed with the substitution of the instance execution currently on top of the stack.
2. The topmost plan instance execution does not verify its invariant or its current state is not a leaf, and there are no more states reachable from it: the plan instance fails. The substitution of the execution is applied to the

plan failure actions which can be executed thereafter; finally, the set of failed instances is updated with the instance at the stack base and all the elements of the stack are removed. New alternatives for the main desire which the intention tried to achieve are then attempted.

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. The action may be:
   (a) `add(BeliefFormula)` or `remove(BeliefFormula)`: `BeliefFormula` (which, at the time of the operation call, must be ground) is added (removed) to (from) the agent's belief set. The agent sends a message to itself notifying that the belief set changed.
   (b) `send(ReceiverAgentId, Message)`: `agentId(AgentId)` is used to find the sender's identity and `sendOp(ReceiverAgentId, AgentId, Message)` is used to send the message.
   (c) `query(SituationFormula)`: if `SituationFormula` instantiated with the current execution substitution can be grounded, and the ground instance is a logical consequence of the agent's beliefs, then the execution substitution is updated by composing it with the grounding substitution. Otherwise, the selected successive state is removed from the set of states reachable from the current state.
   (d) `achieve(Desire)`: if `Desire` instantiated with the current execution substitution can be grounded, and the ground instance is a logical consequence of the agent's beliefs, then the execution substitution is updated by composing it with the grounding substitution. Otherwise, the relevant instances retrieval mechanism is started with the desire instantiated with the current execution substitution.

   After the action has been executed, the current plan instance execution is updated.

6 Coo-BDI at work

To show the advantages of our Coo-BDI extension, we discuss the simple example introduced in Section 1. We want to model the following situation:

1. Mary and Thomas receive an e-mail from John asking to reach him at Liza's home for a party at 7 p.m.
2. Mary would like to join the party: she desires to cancel all previous appointments for this evening and to reach Liza's place at 7 p.m. Instead, Thomas doesn't like parties: he desires to phone Liza at home and decline her invitation.
3. Mary knows how to cancel previous appointments.
4. Mary does not know how to reach Liza's place. Thomas does not how to phone Liza at home.

For each of the four assertions above, we will show how it can be modeled in both BDI and Coo-BDI. We will then discuss the behavior of BDI and Coo-BDI interpreters on the given agent model.

Mary and Thomas receive an e-mail from John. Both in BDI and Coo-BDI, the reception of an e-mail from John can be modeled by the presence of an event

```
received(john,
   message(by(e-mail), content(join_party, liza_home, today, 7pm)))
```

in the event queues of Mary and Thomas.

Mary desires to cancel previous appointments and reach Liza's place. Thomas desires to phone Liza and decline. In BDI there are no clear means to model a mapping between external events and related desires. This mapping can be modeled in an implicit way by defining a plan whose trigger is the external event and whose body contains the desires to achieve. BDI plans are represented by `plan(Trigger, Precondition, Body, Invariant, SuccessActionSequence, FailureActionSequence)`. For sake of readability, throughout the example we will represent plan bodies as drawn trees. In our example, the BDI plan for Mary could be:

```
plan(received(Sender, message(by(CommunicationMeans),
      content(join_party, Where, Day, Hour))),   ⇐ Trigger
   true,                                         ⇐ Precondition
   s0 •
     ↓ achieve(del_appointments(Day, Hour))
   s1 •                                          ⇐ Body
     ↓ achieve(go_to(Where, Day, Hour))
   s2 •,
   true, emptySeq, emptySeq)                     ⇐ Inv., Succ. and Fail.
```

A similar plan, with a different body, could be used to simulate Thomas' generation of desires.

In Coo-BDI mapping between external events and desires is explicitly represented by the `getDesire(OrdinaryEvent, DesireSet)` predicate which every agent must define. The specification of agent Mary includes:

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(del_appointments(Day, Hour),
   achieve(go_to(Where, Day, Hour))})
```

while the specification of agent Thomas includes

```
getDesires(received(Sender,message(by(CommunicationMeans),
   content(join_party, Where, Day, Hour))),
   {achieve(phone(Where, content(decline))})
```

Once the main desires related to John's email reception have been found, one intention is created for each of them. We identify the intention for the `achieve(go_to(liza_home, today, 7pm))` desire by `go_to_liza_intention`.

Mary knows how to cancel previous appointments. Both in BDI and Coo-BDI, the fact that Mary knows how to cancel previous appointments means that at least one plan triggered by `achieve(del_appointments(today, 7pm))` belongs to Mary's plan library. For example, the BDI plan shown below models the fact that deleting appointments with a list of persons requires that the list of persons is retrieved and a message is sent to each person in the list. A plan for managing `achieve(get_appointment_list(Day, Hour, List))` should belong to Mary's plan library, and the external action `multicast(List, content(delete_appointment))` should be defined in terms of `send`.

The Coo-BDI plan could look exactly as the BDI plan plus the access specifier.

```
plan(achieve(del_appointments(Day, Hour)),          ⇐ Trigger
   true,                                            ⇐ Precondition
   s0 •
      ↓ achieve(get_appointment_list(Day, Hour, List))
   s1 •                                                         ⇐ Body
      ↓ multicast(List, content(delete_appointment))
   s2 •,
   true, emptySeq, emptySeq)                        ⇐ Inv., Succ. and Fail.
```

Mary does not know how to reach Liza's place. Thomas does not know how to phone Liza at home. Both in BDI and Coo-BDI this situation is modeled by the absence, in Mary's plan library, of plans triggered by the `achieve(go_to(liza_home, today, 7pm))` desire and by the absence, in Thomas plan library, of plans triggered by the `achieve(phone(Where, content(decline))` desire.

The difference between BDI and Coo-BDI lies in the way the lack of relevant plans for achieving these desires are managed.

Let us consider Mary's case first. In most BDI systems, Mary can drop this desire or she can adopt a default plan. Dropping the desire does not correctly model the usual human behavior: none gives up joining a party just because she/he does not know how to reach the party's place! Using a default plan may be a better solution, but default plans should be carefully designed to be useful in all the situations where no specific plans are available. This requires some intuition to guess all these situations in advance. For example, a default plan saying "search through a white/yellow pages server" may be useful in many situations, including the one of our running example, but it is completely useless to achieve a "prepare a cake" or "pass an exam" desire.

Extending the BDI model with cooperation helps overcoming the lack of relevant plans in an intuitive and human-like way. Let us suppose that Mary's cooperation strategy is characterized by:

```
trustedAgents({john, pattie, michelle})
retrievalPolicy(noLocal)
acquisitionPolicy(add)
```

Since Mary has no local relevant plans for `achieve(go_to(liza_home, today, 7pm))` desire, the retrieval of external plans starts:

– A request request(go_to_liza_request, mary, achieve(go_to(liza_home, today, 7pm))) to provide relevant plans for the desire is created.
– The request is issued to each agent in {john, pattie, michelle} by means of the multicastRequestOp primitive.
– The WaitingOnAgent component of the intention go_to_liza_intention is set to {john, pattie, michelle}.
– The intention go_to_liza_intention is suspended.

The cooperative plan retrieval process goes on according to the content of John's, Pattie's and Michelle's plan library.

John has the only relevant public plan shown below. When he receives Mary's request, he delivers the corresponding plan instance to Mary.

```
plan(public,                                          ⇐ Access specifier
   achieve(go_to(liza_home, Day, Hour)),              ⇐ Trigger
   true,                                              ⇐ Precondition
   s0 •
     ↓ achieve(drive_to(collins_street, Day, (Hour-30min)))
   s1 •                                                         ⇐ Body
     ↓ achieve(park_near(main_church))
   s2 •,
   true, emptySeq, emptySeq)                          ⇐ Inv., Succ. and Fail.
```

Pattie has only a private plan for reaching Liza's place, therefore she sends to Mary an empty set of instances.

Finally, Michelle has the unique relevant plan shown below. Since Mary is included among the allowed recipients of the plan, Michelle sends the corresponding instance to Mary.

```
plan(only({mary, paul}),                              ⇐ Access specifier
   achieve(go_to(liza_home, Day, Hour)),              ⇐ Trigger
   true,                                              ⇐ Precondition
   s0 •
     ↓ achieve(take_a_bus(number(168), Day, (Hour-40min)))
   s1 •                                                         ⇐ Body
     ↓ achieve(get_off(collins))
   s2 •,
   true, emptySeq, emptySeq)                          ⇐ Inv., Succ. and Fail.
```

When Mary receives all the three answers, she does the following:

– She realizes that the intention go_to_liza_int can be resumed.
– She generates all the applicable plans from the set of relevant ones (the two plans received from John and Michelle plus all default plan instances). In this simple example, all relevant plans are applicable as well.
– She selects one of the specific applicable plans associated with the intention.
– Since the chosen plan is an external one (the only specific applicable plans are the ones received from John and Michelle), she records it.

Probably, a human being would behave in a similar way: she/he would contact some friends asking for advice, she/he would select the more convenient instructions and she/he would record the instructions for future reuse.

Thomas will behave like Mary to retrieve the phone number of Liza's home and decline. If we did not decouple events from desires, the completely different attitudes of Mary and Thomas towards parties could not be easily captured. Let us suppose that the reception of the event `received(john, message(by(e-mail), content(join_party, liza_home, today, 7pm)))` triggers a plan like in the BDI setting, and that neither Mary nor Thomas posses a relevant plan for it. The cooperative strategy we implement in Coo-BDI would result into asking plans for managing such an event to all trusted agents. But this makes little sense, since the retrieved relevant plans would represent really heterogeneous suggestions including declining, reaching the party, asking to postpone the party, or quickly inventing an excuse. How could an agent select the right plan among these, if it does not even know which state it desires to achieve? Associating external events with internally generated desires, and cooperating for achieving desires rather than for managing external events overcome this problem.

7 Related and Future Work

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, [18] extends the theoretical BDI framework with the notion of *capability* and investigates how capabilities affect the agent's reasoning about intentions. The BOID (Beliefs, *Obligations*, Intentions, Desires) architecture [3] 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 [7], classical BDI agents are extended with *conditional mental attitudes* represented by interconnected components. The paper [20] investigates the impact that *sociality* has on mental states and how it can be formalized within a BDI model, while [19] focuses on actions and their formalization by adding three more operators to the basic BDI ones: *capabilities*, *opportunities* and *results*. An approach to social reasoning that integrates prior work on *norms* and *obligations* with the BDI approach is discussed in [8]. A generalization of the BDI logic which combines *observation* (formally defined as a bisimulation between the real world and an agent's mental model) and *expectation* (the images of the world in the agent's mind that are associated with observations as about to happen) is proposed in [25].

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 [5] 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. A re-implementation of TOMAS within a Java 2 Enterprise Edition application server is outlined in [4]. It can be used to develop session beans and demonstrates how the distributed transaction model nicely fits into a state-of-the-art environment for mission critical systems in domains such as e-business and Web services. The JAM BDI-theoretic mobile agent architecture [11] provides plan and procedural representations, meta-level 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.

The paper [16] extends the operational semantics of AgentSpeak(L) by giving semantics to speech-act based messages received by an AgentSpeak(L) agent. AgentSpeak(L) is extended to include a mailbox where messages, of the form ⟨`IllocutionaryForce, SenderId, Content`⟩, are kept. Three kind of messages are considered: ⟨`tell, SenderId, Atom`⟩, meaning that `Atom` is in the knowledge base of `SenderId`; ⟨`achieve, SenderId, Atom`⟩ meaning that `SenderId` requests that the recipient of the message achieves a state where `Atom` is true; and ⟨`tellHow, SenderId, Plan`⟩ meaning that `SenderId` uses the plan `Plan` to deal with the event given by `Plan`'s triggering event.

The illocutionary forces (or performatives) of each of them have a dual form (`untell`, `unachieve`, `untellHow`) whose informal meaning is "revert the effect of the previously sent message". While `tell`, `untell`, `achieve` and `unachieve` are standard KQML [14] performatives, `tellHow` and `untellHow` are new ones.

The definition of an agent Ag is also extended to keep track of the sources of the beliefs of Ag, with the set of the agents that Ag trusts and with the set of agents that have some power on Ag.

With respect to other works we have considered in this section, [16] is the work closest to ours. In fact, the extension to AgentSpeak(L) is detailed enough to be implemented. The same holds for our extension. Also, the extension to AgentSpeak(L) allows agents to cooperate by exchanging information (`tell`, `untell`) and by delegating some tasks to other agents (`achieve`, `unachieve`). Our extension has the same aim of enforcing cooperation among agents.

The most significant similarity between the extended AgentSpeak(L) and Coo-BDI, however, lies in the fact that both of them allow agents to exchange plans. In the extension to AgentSpeak(L), plan exchange is realized by means of the ⟨`tellHow, SenderId, Plan`⟩ message. In Coo-BDI it is realized by means of the `provided(SenderId, Request, RelevantInstances)` message. The semantics of ⟨`tellHow, SenderId, Plan`⟩ is that, if the source is trusted, the plan is added to the plan library. The semantics of `provided(SenderId, Request, RelevantInstances)` is that if `Request` is still under management, and one of the applicable `RelevantInstances` is selected to manage the desire inside `Request`, the plan associated to the selected instance is added, discarded or used to replace existing plans according to the agent's acquisition policy.

In both cases, a "basic" BDI agent can be easily modeled. In the extension to AgentSpeak(L), when the set of trusted and powered agents are empty, the agent may simply discard information and requests. 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.

The extensions proposed in [16] and the ones proposed in our paper are so close that it is natural to think they should converge into a unified architecture for highly cooperative BDI agents. Messages should adhere to the form proposed in [16], so that they have a clear semantics. Plans should be extended with access specifiers, so that the agents can decide when a plan should be shared with other agents by means of a `tellHow` message. The interpreter should take both extensions into account. It should be implemented to get feedbacks on the formalization, (possibly) confirming the adequacy of the approach presented in both papers. We are currently working with the authors of [16] to realize this extension.

Acknowledgments. The authors thank Rafael Bordini, Paolo Busetta, 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

1. R. H. Bordini, A. L. C. Bazzan, R. de 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 the 1st International AAMAS Joint Conference*, pages 1294–1302. ACM Press, 2002.
2. R. H. Bordini and Á. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In J. Dix, J. A. Leite, and K. Satoh, editors, *Proc. of the 3rd International CLIMA Workshop. Electronic Notes in Theoretical Computer Science 70(5)*. Elsevier, 2002.
3. J. Broersen, M. Dastani, J. Hulstijn, Z. Huang, and L. van der Torre. The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In *Proc. of the 5th International Agents Conference*, pages 9–16. ACM Press, 2001.
4. P. Busetta, J. Bailey, and R. Kotagiri. A reliable computational model for BDI agents. In *Proc. of the Workshop on Safe Agents. Held in conjuction with AAMAS'03*, 2003.
5. P. Busetta and K. Ramamohanarao. An architecture for mobile BDI agents. In *Proc. of the 1998 ACM SAC Symposium*, 1998.
6. P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents – components for intelligent agents in Java. *AgentLink News Letter*, 2, 1999.
7. M. Dastani and L. van der Torre. An extension of BDI_{CTL} with functional dependencies and components. In M. Baaz and A. Voronkov, editors, *Proc. of the 9th International LPAR Conference*, pages 115–129. Springer Verlag, 2002.
8. F. Dignum, D. Morley, E. Sonenberg, and L. Cavendon. Towards socially sophisticated BDI agents. In *Proc. of the 4th International ICMAS Conference*, pages 111–118. IEEE Computer Society, 2000.

9. M. d'Inverno, K. V. Hindriks, and M. Luck. A formal architecture for the 3APL agent programming language. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *Proc. of the 1st International ZB Conference*, pages 168–187. Springer Verlag, 2000. LNCS 1878.
10. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. Wooldridge, editors, *Proc. of the 4th International ATAL Workshop*, pages 155–176. Springer Verlag, 1997. LNAI 1365.
11. M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. *AgentLink News Letter*, 5, 2000.
12. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.
13. R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Proc. of the LID International Workshop*, pages 137–149. Springer Verlag, 1996.
14. J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Proc. of the 2nd International ATAL Workshop*, pages 347–360. Springer Verlag, 1995.
15. Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In *Proc. of the LABS Workshop, held in conjunction with KR'02*, pages 45–59, 2002.
16. 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. 2003. In this volume.
17. K. L. Myers. User guide for the procedural reasoning system. Technical report, Artificial Intelligence Center, Menlo Park, CA, 1997.
18. L. Padgham and P. Lambrix. Agent capabilities: Extending BDI theory. In *Proc. of the 7th AAAI National Conference*, pages 68–73. AAAI Press–The MIT Press, 2000.
19. V. Padmanabhan, G. Governatori, and A. Sattar. Actions made explicit in BDI. In M. Stumptner, D. Corbett, and M. J. Brooks, editors, *Proc. of the 14th Australian Joint Conference on AI*, pages 390–401. Springer Verlag, 2001.
20. P. Panzarasa, T. Norman, and N. R. Jennings. Modeling sociality in the BDI framework. In J. Liu and N. Zhong, editors, *Proc. of the 1st Asia-Pacific Conference on Intelligent Agent Technology*, pages 202–206. World Scientific Publishing, 1999.
21. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer Verlag, 1996. LNAI 1038.
22. A. S. Rao and M. Georgeff. BDI agents: from theory to practice. In V. Lesser and L. Gasser, editors, *Proc. of the 1st International ICMAS Conference)*, pages 312–319, 1995.
23. A. S. Rao and M. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293 – 342, 1998.
24. J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.
25. B. V. Trân, J. Harland, and M. Hamilton. A combined logic of expectation and observation (a generalization of BDI logics). 2003. In this volume.
26. M. Winikoff. AgentTalk Home Page. `http://goanna.cs.rmit.edu.au/~winikoff/agenttalk`, 2001.
27. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.