Interaction among agents that plan

Felipe Rech Meneguzzi
King's College London
Department of Computer Science
London, United Kingdom
felipe.meneguzzi@kcl.ac.uk

Michael Luck
King's College London
Department of Computer Science
London, United Kingdom
michael.luck@kcl.ac.uk

ABSTRACT

The development of practical agent languages has progressed significantly over recent years, but this has largely been independent of distinct developments in aspects of multiagent cooperation and planning. For example, while the popular AgentSpeak(L) has had various extensions and improvements proposed, it still essentially a single-agent language. In response, in this paper, we describe a simple, yet effective, technique for multiagent planning that enables an agent to take advantage of cooperating agents in a society. In particular, we build on a technique that enables new plans to be added to a plan library through the invocation of an external planning component, and extend it to include the construction of plans involving the chaining of subplans of others. Our mechanism makes use of *plan patterns* that insulate the planning process from the resulting distributed aspects of plan execution through local *proxy plans* that encode information about the preconditions and effects of the *external plans* provided by agents willing to cooperate. In this way, we allow an agent to discover new ways of achieving its goals through local planning and the delegation of tasks for execution by others, allowing it to overcome individual limitations.

Categories and Subject Descriptors

D.2.5 [**Artificial Intelligence**]: Programming Languages and Software; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multi-agent systems*

General Terms

Planning, Interaction

Keywords

Agent Languages, BDI, Cooperation, Planning

1. INTRODUCTION

Agent-based software has been advocated as an ideal technique for the development of large, distributed applications, viewing them as a number of independently controlled parts that interact and cooperate to achieve their design objectives. The agent model perhaps most commonly used in the development of agent-oriented programming languages is based on the mental states of beliefs, desires and intentions, or BDI. In real-world scenarios, BDI-type agents often have a large plan-library to cope with a complex world, and need to include plans to deal with all contingencies foreseen by the designer. In consequence, much research dealing with agent languages has focused on the description of plans used by an *individual* agent to interact with the world. Although in multiagent systems, agents are assumed to be able to use interaction to achieve goals, agent languages seldom provide mechanisms to do so, and cooperation is generally developed in an *ad hoc* fashion. Even when cooperation is involved, it tends to use a highly specialised version of any of a number of existing cooperation techniques, assuming a distributed but ultimately *predefined* set of abilities in the society.

Previous work has shown the applicability of planning algorithms in the generation of new individual plans through composition of existing plans in a plan library [12]. When a planning-capable agent needs to achieve a new goal, it searches its plan library for applicable plans. If no suitable plan is found in the plan library, the planner is invoked in an attempt to generate a new plan to satisfy the desired goal. Plans generated by the planner in this way are added to the plan library and become available to the agent to solve future instances of the particular goal. However, if the planner fails to generate a new plan, this (normally) means that the proposed goal is impossible to achieve, given the world state and capabilities known to the agent at the time of planning. Such an approach to planning has been studied for individual agents, planning over their own capabilities, under the premise that these capabilities are static throughout its life cycle.

However, in a multi-agent environment the limitations of an individual agent may be overcome with the help of others, either by delegating tasks to other agents, or by learning new ways of achieving goals. This means that the assumption that capabilities are static no longer holds. In this case, a failure to generate a plan from an individual's capabilities does not necessarily mean the goal is impossible, since other agents in the society might have complementary capabilities. If an agent is capable of generating new plans at runtime by taking into consideration the capabilities of others, new *multiagent* plans can be used to overcome individual limitations, and can also be added to the plan library for future use. In a simplistic approach, a planning capable agent interpreter can be used to achieve this effect.

Cooperative action involves communication and coordination, as well as an increased degree of risk for the success of a plan, given that the agents relied upon may break their commitments to achieve their own goals. But from the planner's perspective, the composition of new plans based on preconditions and effects upon the environment can be performed independently of these factors. It is important to point out that, though there are a number of issues that must be addressed in order to perform planning in distributed systems [4], communication and coordination can be abstracted away from the planning process and inserted at a later point in time [9]. In this paper we develop a new technique for agents with plan generation capabilities to cooperate in a multiagent society. In particular, our technique relies on plan patterns (described in Section 3.2) that encapsulate communication and coordination in such a way that the planning algorithm can ignore them when chaining operators in a plan. Though many existing approaches to cooperative action handle communication and coordination together with plan composition [11], we choose to separate these two tasks in order to allow the use of *off-the-shelf* planning algorithms. The rationale behind this choice of approach is as follows:

- there is a wide range of local planning algorithms;
- active research on planning algorithms yields new potentially useful algorithms;
- some planning algorithms are better suited for certain specific domains;
- integrating communication and cooperation in the planning algorithm is not always easy; and
- our approach delegates actions to the agent architecture, allowing new planners to be used seamlessly.

Even if the issues arising from interaction can be ignored by the planner, they must be addressed in order to ensure the long-term effectiveness of the plan library. In particular, relying on third parties to accomplish one's goals can be a problem due to unreliability and broken commitments. Moreover, it is possible that an agent whose cooperation is necessary for some plan in the plan library may leave the society. This renders such a plan not only useless, but also damaging to an agent's efficiency if the plan is eventually selected to achieve some goal, since this plan will always require the agent to drop it after wasting the effort of starting to execute it. Therefore, we provide a mechanism to *clean up* cooperative plans when they become obsolete. By extracting key information from other agents' plans, particularly in relation to the declarative consequences of local plans, an agent can be informed of the problem-solving capabilities of others, allowing it to delegate the achievement of specific world-states, and using this information in its own planning process.

Our contribution in this paper is twofold: a generic technique to reduce multiagent planning into a traditional planning problem, and the practical integration of such a technique in a BDI-like agent language. In our approach, external plans are encapsulated into patterns of local plans in order to abstract the communication and coordination aspects away from the planner.

The paper is organised as follows: in Section 2 we summarise the background work upon which our own contribution is based, reviewing AgentSpeak(L) and AgentSpeak(PL); we then proceed to explain in detail our multiagent planning technique in Section 3; followed by a discussion of related work in Section 4; finally, in Section 5, we summarise and conclude.

2. BACKGROUND

2.1 Cooperation

Cooperation is often cited as one of the main characteristic properties of multiagent systems [7, 6]. Yet there are several different modes of cooperation that can be identified: *i)* multiple agents acting towards a common joint goal; *ii)* one agent acting to achieve goals for another agent; and *iii)* agents synchronising their actions so as to avoid negative interference.

The first, and most common mode of cooperation in agents consists of a group of agents sharing a possibly implicit joint goal and acting to achieve this goal in a coordinated way. This goal might be negotiated at runtime or exist in all agents by design. The second possible mode of cooperation consists of one or more agents performing actions that are not directly related to their own goals, but rather support the achievement of the goals of another agent. The third and final mode of cooperation commonly considered consists of agents agreeing on some coordination of their individual actions towards their individual goals in such a way that no agent jeopardises the operation of another. Because our starting point is one individual agent, seeking to achieve its goals through the assistance of another, our focus in this paper is on the second mode of cooperation.

2.2 AgentSpeak(L)

AgentSpeak(L) [16] is an agent language, as well as an abstract interpreter for the language, and follows the *beliefs, desires and intentions* (BDI) model of practical reasoning. In simple terms, a BDI agent tries to realise the *desires* it *believes* are possible by committing to carrying out certain courses of action through *intentions*. The language of AgentSpeak(L) allows the definition of *reactive procedural plans*, so that plans are defined in terms of events to which an agent should react to by executing a sequence of steps (*i.e.* a procedure). Plan execution is further constrained by the context in which these plans are relevant. Here, a plan is executed under the assumption that some implicit goal is being accomplished by the plan at the particular moment.

The control cycle of an AgentSpeak(L) interpreter adopts plans in reaction to events in the environment and executes their steps. If the step is an action it is executed, while if the step is a goal, a new plan for the goal is added into the intention structure. Failures may take place either in the execution of actions, or during the processing of subplans. When such a failure takes place, the plan that is currently being processed also fails. Thus, if a plan selected for the achievement of a given goal fails, the default behaviour of an AgentSpeak(L) agent is to conclude that the goal that caused the plan to be adopted is not achievable. This control cycle[1] strongly couples plan execution to goal achievement.

In order to better understand the relationship between the control cycle and the plan library, it is necessary to introduce the notation of AgentSpeak(L) plans. Events on an agent's data structures that can trigger the adoption of plans consist of additions and deletions of goals and beliefs, and are represented by the plus $(+)$ and minus $(-)$ sign respectively. Goals are distinguished into *test goals* and *achievement goals*, denoted by a preceding question mark (?), or an exclamation mark (!), respectively. For example, the addition of a goal to achieve g is represented by $+!g$. Belief additions and deletions arise as the agent perceives the environment, and are therefore outside its control, while goal additions and deletions only arise as part of the execution of an agent's plans. Plans in AgentSpeak(L) are represented by a header comprising a triggering condition and a context, as well as a body describing the steps the agent takes when a plan is selected for execution as is illustrated in Figure 1. If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then $e : b_1 \& \ldots \& b_m \leftarrow h_1; \ldots; h_n.$ is a plan. As an example, consider a plan associated with the triggering event $!move(O, A, B)$ corresponding to the goal of moving an object O from A to B, where:

[1]For a full description of AgentSpeak(L), see d'Inverno *et al.* [5]

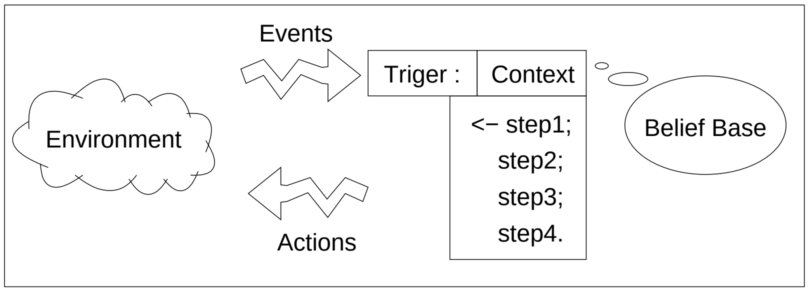

Figure 1: AgentSpeak(L) plan and dynamics.

- *e* is `!move(O,A,B)`;
- `at(O,A)` and **`not`** `at(O,B)` are belief literals; and
- `-at(O,A)` and `+at(O,B)` are two steps in the plan body, consisting of information about belief additions and deletions.

The plan is then as follows:

```
+!move(O,A,B) : at(O,A) & not at(O,B)
    <- -at(O,A);
       +at(O,B).
```

When this plan is executed, it should result in the agent believing that O is no longer in position A, and then believing it is in position B. For an agent to rationally want to move O from A to B, it must believe O is at position A and not already at position B.

2.3 Planning in AgentSpeak(PL)

AgentSpeak(PL) [12] is an extended AgentSpeak(L) interpreter that integrates a planning module capable of generating new high-level plans by chaining lower-level plans in an agent's plan library. Planning in AgentSpeak(PL) relies on a process that extracts information about the declarative *consequences* of simple AgentSpeak(L) plans and uses this information, together with these plans' *context conditions*, to generate equivalent STRIPS-like planning operators. Here, we review the relevant aspects of AgentSpeak(PL), which we later use in the description of our multiagent technique.

The design of a traditional AgentSpeak(L) plan library follows a similar approach to programming in procedural languages, where a designer typically defines fine-grained actions to be the building blocks of more complex operations. These building blocks are then assembled into higher-level procedures to accomplish the main goals of a system. Analogously, an AgentSpeak(L) designer traditionally creates fine-grained *plans* to be the building blocks of more complex operations, typically defining more than one plan to satisfy the same goal (*i.e.* sharing the same trigger condition), while specifying the situations in which it is applicable through the *context* of each plan. Here, STRIPS actions are likened to low-level AgentSpeak(L) *plans*, since the effects of primitive AgentSpeak(L) actions are not explicitly defined in an agent description.

Once the building-block procedures are defined, higher-level operations must be defined to fulfil the broader goals of a system by combining these building blocks. In a traditional AgentSpeak(L) plan library, higher-level plans to achieve broader goals contain a series of goals to be achieved by the lower-level operations. This construction of higher-level plans that make use of lower-level ones is analogous to the planning performed by a propositional planning system. By doing the *planning themselves*, *designers* must cope with every foreseeable situation the agent might find itself in, and generate higher-level plans combining lower-level tasks accordingly. Moreover, the designer must make sure that the subplans being used do not lead to conflicting situations. In AgentSpeak(PL), by contract, this responsibility is delegated to a STRIPS planner.

Plans resulting from propositional planning can then be converted into sequences of AgentSpeak achievement goals to comprise the body of new plans available within an agent's plan library. Here, an agent can still have high-level plans pre-defined by the designer, so that routine tasks can be handled exactly as intended. At the same time, if an unforseen situation presents itself to the agent, it has the flexibility of finding novel ways to solve problems, while augmenting the agent's plan library in the process.

3. PLANNING AND COOPERATION

As we have seen, when an agent has exhausted its individual options to achieve a goal, it may be able to accomplish this goal through others. In order to generate new plans that rely on cooperation with others, we define a practical strategy for multi-agent planning and cooperation that allows an agent to share the knowledge of the consequences of its plans so that others can delegate parts of their high-level plans and achieve new goals. We introduce *external plans*, which are plans *owned* by one agent, whose declarative consequences are known by others and can, therefore, be requested by others to help achieve their aims. Newly constructed external plans can be integrated into multi-agent plans generated through classical planning problems by considering their preconditions and consequences and equating them to STRIPS/PDDL operators, as described in Section 2.3. These newly created plans are then integrated into an agent's plan library for future use and efficiency gains. Furthermore, our strategy takes into consideration the unreliability of cooperation in the context of self-interested and unreliable agents by associating failure handling plans to manage multi-agent plans, and eventually to remove plans that include such unreliable partners.

In more detail, given an agent with plans it is willing to execute on behalf of others (*i.e.* its *shared plans*), our technique consists of automatically generating plans in both the sharer's plan library and the requester's plan library using reusable *plan patterns*. These new plans encode all the communication necessary for a requesting agent to delegate the achievement of the external plan, and encapsulate information about the declarative effects of such an external plan, allowing a planning module on the requester's side to use these plans in newly created plans. Moreover, we use plan patterns to generate failure handling plans to cope with the potential unreliability of the sharer.

In this section we define three plan patterns that generate new plans based on an existing plan an agent is willing to execute on

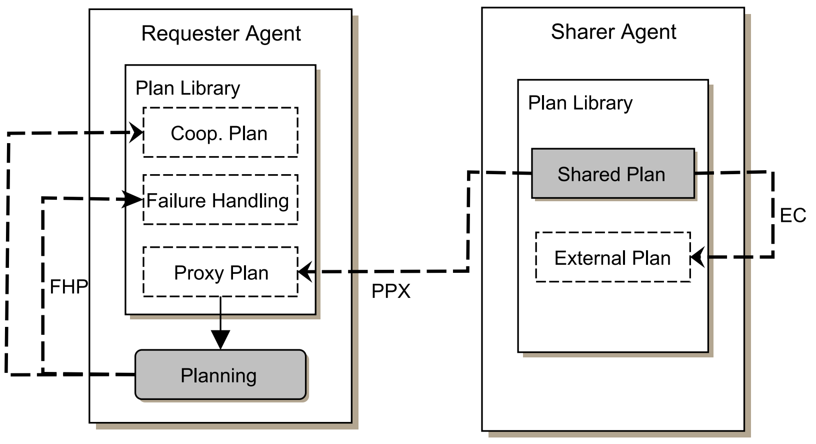

Figure 2: Plan patterns involved in the sharing and use of a plan.

behalf of others, generating the necessary framework for a form of cooperation based on delegation without the need for the designer to predefine cooperative plans. More specifically, given a *shared plan*, we define an *external plan* (EC) pattern that includes the steps necessary for another agent to request the execution of the shared plan. On the requester's side, we define a *proxy plan* (PPX) pattern that encodes the declarative information of the shared plan's preconditions and consequences, and contains the steps necessary for the requester to request the remote execution of the shared plan. Ultimately, proxy plans can be used by the planning process of AgentSpeak(PL) as if they were local plans, to provide new *cooperative plans*, but given the uncertain nature of agent cooperation, there is also a need to provide *failure handling plans* (FHP) to cope with unreliable partners. These patterns and their resulting plans (*i.e.* the plans that are generated from the plan pattern) are summarised in the diagram of Figure 2, where dashed arrows represent the creation of new plans through a plan pattern.

3.1 Communication for Cooperation in AgentSpeak

Our technique assumes a BDI-style language [16] with a construct for declarative goals, and speech-act based communication. We also assume two other language features: the ability to *annotate plans* with additional information; and the notion of *internal actions*. Examples of agent languages suitable for implementing this strategy are CANPLAN2 [17] and Jason [3]. Descriptions of agent plans throughout this section use Jason, but these plan definitions can be easily converted to any BDI-like agent language. Jason is a recently developed AgentSpeak(L) interpreter with a number of extensions necessary for our technique to function in practice. Here, we summarise the language features we use in the descriptions throughout this section, giving notation details when relevant.

3.1.1 Internal actions

The common understanding of agent actions is that they are environment transformation operators, so that when an agent invokes an action, some consequence in the environment is expected. However, when some custom computation needs to take place within a single reasoning cycle, Bordini *et al.* use the concept of an *internal action* in AgentSpeak(XL) [2]. This allows an agent to access extensible libraries of custom procedures that can be executed instantaneously by an agent. Unlike traditional actions, internal actions do not cause changes in the environment, and since they are executed instantly, they can be included in either the body or the context of a plan, to refine the process of selecting applicable plans. Syntactically, internal actions are denoted in the language by a preceding "." character, so the invocation of a *check* internal action with two parameters is represented as $.check(a, b)$.

3.1.2 Speech-act based communication

Effective cooperation between autonomous agents requires some form of communication, typically using an agent communication language, such as FIPA or KQML [19]. From an agent language perspective, Moreira *et al.* [13] have introduced an operational semantics of speech-act based communication for AgentSpeak(L), defining plan rules for handling several of the performatives defined by Searle [18]. These plan rules are given from both a sender and a receiver point of view, allowing them to be implemented in practical AgentSpeak(L) interpreters. In this paper we are concerned with three performatives:

- *ask*, used by an agent to request information from others;
- *tell*, used by an agent to supply information to others; and
- *achieve*, used by an agent to request another agent to achieve a procedural goal.

From an operational perspective, we consider an implementation of agent message passing using the concept of *internal actions* described above, because messages between agents are not expected otherwise to cause the environment to change. Messages are therefore sent using the $.send$ internal action, which takes three parameters: the identification of the receiver, the performative, and the message content [2]. In terms of representation of beliefs, annotations [2] have been used to provide additional information regarding the source of events from external communication rather than the environment. Thus, if the addition of the belief $time(hockey, 1020)$ by $randall$ is a result of communication from $dante$ rather than a simple perception, the event posted to $randall$ is represented as $time(hockey, 1020)[source(dante)]$, denoting this belief's origin.

[2]In this paper we provide a simplified overview of how these performatives are operationalised in AgentSpeak(L), overlooking a number of details regarding cooperation policies, and more complex handling of communication-related event processing by AgentSpeak(L). For additional information on these details, consult [13].

When an agent sends a message with an *ask* performative, it wants to ascertain that some expression unifies with another's belief base. For example, suppose agent *randall* wants to know the time of the hockey game, stored in the belief base of agent *dante* as the belief $time(hockey, T)$. To discover this information, it executes an internal action $.send(dante, ask, time(hockey, T))$, which causes an event $+?time(hockey, T)$ to be posted to *dante*. If *dante* accepts the message, and has $time(hockey, 1020)$ in its belief base, the $.send$ action in *randall* is executed successfully, resulting in T being unified with 1020. Notice that since the effect of this send is an event in the receiving agent, it might be handled by a plan with a triggering event matching the query being made to it, rather than a direct query to its belief base.

Similarly, when an agent sends a message with a *tell* performative, it wants to make another agent aware of some belief expression. Now suppose *dante* wants to make *randall* aware that the hockey game is at 1020 by executing the action $.send(randall, tell, time(hockey, 1020))$. If *randall* accepts this message, it causes the event $+time(hockey, 1020)$ to be posted to *randall*.

Finally, when an agent wants another agent to adopt a particular achievement goal, it sends a message with an *achieve* performative. So if *dante* wants *randall* to come to the hockey game now, and it knows that *randall* has a plan to come to the game associated with the triggering event $+!comeToHockey$, it executes $.send(randall, achieve, comeToHockey)$. Again, if *randall* accepts this message, $+!comeToHockey$ is posted to *randall*, and the plan it executed.

3.2 A multiagent planning mechanism

When an agent has failed to achieve a goal through its individual capabilities and its previously known cooperative strategies, it engages in multi-agent planning to try and solve the problem with a new cooperative plan. Our technique is divided into three main parts: the discovery of potential cooperation partners, the creation of cooperative plans while abstracting cooperation, and the execution of multiagent plans.

3.2.1 Plan patterns

While many researchers have chosen to create new languages to add notions such as declarative goals [21, 3] and failure handling mechanisms [20, 17], it is possible to represent these, and many other notions using simpler, existing agent languages. For example, in AgentSpeak(L), all of these notions can be represented by multiple related plans, as shown by Hübner *et al.* [10], who introduce the notion of *plan patterns* to facilitate the designer's task of creating multiple, related, plans that serve a particular purpose.

Here, we consider a plan pattern to be an agent program rewriting rule with a numerator describing the original plan (or plans) description, and a denominator describing the resulting agent program. So, for example, if we wish to define a plan pattern that adds a printed message before and after a certain plan body b is executed, called **PDB** (Plan Debug), the rule is defined as:

$$\frac{+e : c \leftarrow b.}{+e : c \leftarrow .print(\text{"Start"}); b; .print(\text{"End"}).}\,\textbf{PDB}$$

3.2.2 Primitives

In order for an agent to find external plans in a society, it must seek partners willing to carry out plans on behalf of the requesting agent. These willing partners then send declarative information about their plans, that is their preconditions and effects. Here, we are not concerned with the actual mechanism used in the discovery of partners, and plan patterns are meant to be an abstraction of any of a number of existing partner selection mechanisms, such as that described in [14]. In this description of our method, we assume that cooperation partners have already been selected somehow, and our capability discovery method consists of broadcasting a request for external plans, which is answered by all available agents in the society. However, if a partner selection mechanism is in place, the requests for external plans will only be sent by selected cooperation partners.

Partners wishing to inform others of their external plans need to gather the plan invocation parameters, preconditions and declarative effects and send this information to their peers. This information can be retrieved using the same process as in AgentSpeak(PL), but instead of using this information to generate a STRIPS-like operator description, an agent sends this as a reply to another agent requesting external plans, along with the identification of the agent supplying the external plan. This is represented in the tuple $\langle g, a, P, E\rangle$, where:

- g is the achievement goal (including parameters) in the sharing agent's plan library that will be adopted on behalf of the requesting agent;
- a is the identifier of the sharing agent that owns the external plan;
- P is a set $\{p_0, \ldots, p_n\}$ of preconditions of g; and
- E is a set $\{r_0, \ldots, r_m\}$ of declarative effects expected to hold after the external plan is executed.

This information is used in the creation of plan patterns that serve as local placeholders for the invocation of externally executed plans, which we call *proxy plans*. The creation of proxy plans is detailed next.

3.2.3 Creating proxy plans

Once an agent is aware of the external plans of others in the same environment, it can try to use these capabilities in its own problem-solving. In this approach, we make the *external* aspect of *shared plans* transparent to an agent's local planner through *proxy plans*. These proxy plans describe the expected outcome of a successful invocation of a third party capability and encapsulate the communication and coordination necessary for effective cooperation. A proxy plan pattern **PPX** for an external plan $\langle g, a, P, E\rangle$, where $P = \{p_0, \ldots, p_n\}$, and $E = \{b_0, \ldots, b_m\}$ (and b_i are belief additions or deletions) is:

$$\frac{+!g : p_0 \& \ldots \& p_n \leftarrow b_0; \ldots; b_m}{\begin{array}{rl} +!remoteG : & p_0 \& \ldots \& p_n \& ready(a, g) \\ \leftarrow & .send(a, achieve, requestG); \\ & .wait(done(g)); \\ & b_0; \ldots; b_m. \\ +!check(a, g) : & true \\ \leftarrow & +ready(a, g). \end{array}}\,\textbf{PPX}_{g,a,P,E}$$

This plan pattern creates two plans, one of which replicates all the logical constraints required for a to be successful in executing this plan locally. The plan body includes a communication action ($.send$) that uses the *achieve* performative to request the sharing agent to carry out the specified plan, followed by an action to wait for confirmation that the plan was executed. Finally, the plan pattern replicates the belief additions expressed in the sharing agent's external plan, so that the planning process of AgentSpeak(PL) [12] can process this plan in the same way as it would process local plans.

In addition to the action-related part of the proxy plan to invoke the external plan, one may also want to check that the owner of the external plan is ready and willing to adopt the external plan. This is represented in the **PPX** plan pattern by the precondition $ready(a, g)$, which is added to those preconditions already present in the original external plan, and is the result of an extra plan to ensure that the sharing agent will actually carry out that action when the requesting agent needs it to do so. In the **PPX** plan pattern, this plan is simply a placeholder for any mechanism used to ascertain the reliability of a cooperation partner, which can be replaced by any mechanism preferred by the designer. Such a mechanism can be introduced using a new **CA** (check agent) plan pattern, which rewrites the $check$ plan so that it calls a plan in the plan library associated with this mechanism. For example, if there is a trust verification mechanism associated with a *verifyTrust* achievement goal (which we will not specify, but assume to be specified by the designer), a plan pattern **CA** for the readiness of an agent to execute external plan $\langle g, a, P, E\rangle$ through an achievement goal *verifyTrust* is:

$$\frac{+!check(a,g) : true \leftarrow +ready(a,g).}{\begin{array}{ll} +!check(a,g) : & true \\ \leftarrow & !verifyTrust(a,g); \\ & +ready(a,g). \end{array}}\,\mathbf{CA}_{g,a}$$

3.2.4 Creating external plans

An important property of our proxy plans is that they succeed when the sharer agent succeeds, and fail if either the sharer agent fails in its execution or it refuses to carry out its commitment. Hence, from the requester agent's point of view, the execution of a local plan and an external plan is the same.

Naturally, an agent sharing an external plan needs to have in its plan library the achievement goal that corresponds to the *achieve* performative sent by the requesting agent. We refer to this achievement goal as a *plan endpoint* to the **PPX** plan pattern, which is associated with an actual plan in the sharing agent's plan library. The external plan, therefore, is generated from a local plan in the sharer's plan library using the **EP** (external plan) pattern, which is as follows:

$$\frac{+!g : e \leftarrow b.}{\begin{array}{rl} +!g : e & \leftarrow b. \\ +!requestG[source(S)] & : true \\ \leftarrow & !g; \\ & .send(S, tell, done(g)). \end{array}}\,\mathbf{EP}_{g}$$

3.2.5 Creating cooperative plans

Given the properties of the proxy plans described above, it is easy to use the planning approach of AgentSpeak(PL) to generate new multi-agent plans, since the AgentSpeak(PL) planning module is insulated from the communication and cooperation aspects of planning. However, although the generation of a sequence of actions (from a cause and effect perspective) does not depend directly on whether it includes external and internal capabilities, high-level plans that depend on the compliance of third parties must contain guards to prevent initiating the plan when it has become infeasible. These guards are derived by propagating the preconditions of external proxy plans to the precondition of the high-level plan generated by the planning module. Propagating these preconditions ensures that a plan will not be initiated until all parties are ready to comply

```
+!goal_conj([closed(store)]) : at(randall, store)
     & ready(randall)
   <- !remoteClose(store).
```

Listing 1: A cooperative plan.

with requests for cooperation, while making sure that the cooperating agent is queried for availability just before its cooperation is needed.

As an example, suppose that $dante$ is aware that $randall$ can achieve a goal to close the store on its behalf. If $dante$ needs $randall$ to close the store on its behalf, it requires $randall$ to be at the store, and results in the store being closed; a cooperative plan to achieve these goals generated in our system is shown in Table 1.

When a cooperative plan is adopted by an agent, it eventually reaches the step corresponding to the adoption of the proxy plan ($remoteG$). The proxy plan causes this agent to send a message to the sharer requesting it to execute its external plan ($requestG$), which corresponds to delegating the adoption of a plan to achieve goal g in the sharer's plan library. If the plan to achieve g is executed successfully, the sharer sends confirmation of having achieved g. This sequence of events is illustrated in Figure 3.

3.2.6 Failure handling for new plans

Although the ability to create new plans taking advantage of the external plans of other agents allows the creation of plans that achieve goals otherwise impossible to an agent, the dependence on other self-interested agents poses another challenge, coping with possibly unreliable partners. Plans created at design time tend to be very efficient by making assumptions about aspects of the environment that do not change at runtime, whereas the *generation* of plans at runtime involves a great deal of computational effort. However, plans created in a dynamic society in which autonomous agents may join and leave at any point in time cannot make many assumptions regarding the availability of capabilities shared by third parties. The likelihood of failure for plans that depend on others can, therefore, be considered greater than for plans that rely on an individual's own capabilities. Thus, it is necessary for dynamically generated plans, especially those that depend on unreliable capabilities, to have associated failure handling plans. Here, handling plan failures is important to ensure that an agent can cope with faults due to failed cooperation. It also allows an agent to manage its plan library in the long term, removing plans that are no longer relevant due to the absence of, or consistent lack of reliability of, necessary parties. For example, if an agent creates a plan that involves cooperation with an agent a, we introduce a *failure handling plan* **FHP** pattern that removes the failed plan when a fails to cooperate for some reason, as follows:

$$\frac{+!goal\_conj([g_1,\ldots,g_n]) : e \leftarrow b.}{\begin{array}{ll} +!goal\_conj([g_1,\ldots,g_n]) & : e \leftarrow b. \\ -!goal\_conj([g_1,\ldots,g_n]) & : notready(a) \\ \quad \leftarrow .remove\_plan(goal\_conj([g_1,\ldots,g_n])). & \end{array}}\,\mathbf{FHP}_{a}$$

4. RELATED WORK

Previous work by Ancona *et al.* [1] provides a cooperation technique that allows agents to expand their problem solving capabilities by exchanging plans at runtime. Although this technique relies on a very similar basic agent framework (aside from the planning component), it has a distinct approach to addressing the shortcomings of an agent, as it relies on an agent receiving entire plans from

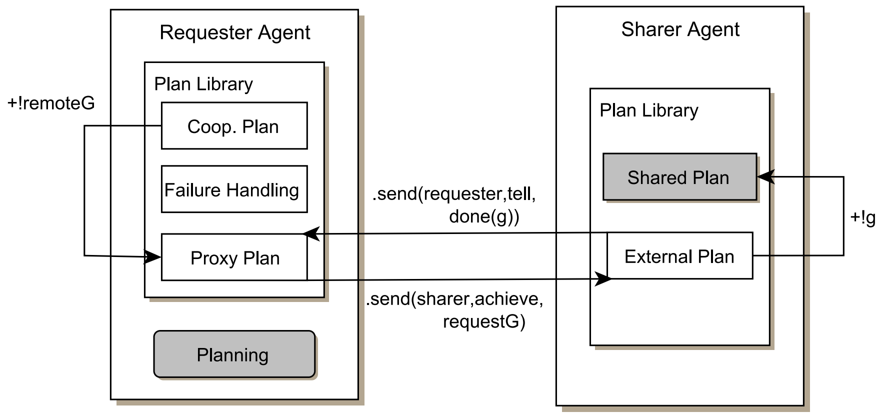

Figure 3: Proxy plan communication.

others. In particular, it assumes that all agents in an environment are able to execute the same set of basic actions, which may not be the case in many real world scenarios. For example, agents might require different levels of authorisation to perform specific actions in the environment: an agent running in a user-level account, doing maintenance in a Unix filesystem may need to change a file that is owned by the root user, and clearly the plans that the root can execute cannot simply be sent to this agent. Ancona's approach is complementary to ours in the sense that it can, for example, replace the planning module we use to generate new plans from scratch and allow an agent to get new plans from others.

It may be argued that creating cooperative plans using preconditions and effects information in AgentSpeak(L) is akin to *Service Oriented Architectures* (SOA), through web services being shared by a directory service accessed through some protocol like the *Universal Description, Discovery and Integration* (UDDI) protocol [15]. Indeed, web services are a possible technology for the instantiation of an agent system using an SOA to provide web-protocols for the communication layer of such a system. Unlike web services on their own, however, agents have intentionality, and do not necessarily carry out the requests of a client. Directory services could also be used in a web service-based implementation, but they add a centralising characteristic that is not entirely necessary for our technique, since the directory service does not take into account the dynamic nature of an agent's willingness to cooperate; that is, an agent *A* may agree to execute an action on behalf of agent *B* at one point in time, but not at another, whereas a service is expected always to respond in the same way.

5. CONCLUSIONS

By taking advantage of recent developments in practical agent languages, we have described a practical, yet flexible, technique for multiagent planning. This technique extends previous work on agent planning [12] to take advantage of the availability of cooperating agents in a society, allowing agents to overcome individual limitations by delegating *parts* of locally generated plans for execution by others. In this paper we have shown how this technique can be implemented using recent extensions to the AgentSpeak(L) language, without affecting the generality of our approach, since any other BDI-like language with declarative goals and communication capabilities can be extended with the planning we propose.

The focus of the paper is on the structural and functional aspects of the plan library, and as a consequence we have sidestepped any detailed account of how to address two major issues with cooperation in agents: the distribution of the planning effort, and the evaluation of reliability of cooperation partners. However, by modularising our technique, a designer can choose from the existing body of work in both these areas. Moreover, we acknowledge that issues of trust and reliability of cooperation partners are of paramount importance in any deployment of a system composed of agents that use our technique, but this is a separate issue, and is isolated from the rest of our planning process. Regarding the issue of distribution, although the classical planning module leveraged from AgentSpeak(PL) [12] is simple and centralised, we see no hurdles in using our technique with distributed plan formation algorithms, such as that proposed by Zhang *et al.* [22]. In this respect, our method is flexible in that it allows any planning algorithm with a PDDL [8] compatible planner to be used in the planning module.

Acknowledgments

The first author is supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) of the Brazilian Ministry of Education.

6. REFERENCES

[1] D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-agentspeak: Cooperation in agentspeak through plan exchange. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 696–705, 2004.

[2] 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 *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302, 2002.

[3] R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

[4] M. E. desJardins, E. H. Durfee, C. L. O. Jr., and M. J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13–22, 1999.

[5] M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

[6] M. d'Inverno, M. Luck, and M. Wooldridge. Cooperation structures. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 600–605, 1997.

[7] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *Knowledge Engineering Review*, 12(3):309–314, 1997.

[8] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[9] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[10] J. F. Hübner, R. H. Bordini, and M. Wooldridge. Programming declarative goals using plan patterns. In M. Baldoni and U. Endriss, editors, *Proceedings of the Fourth Workshop on Declarative Agent Languages and Technologies*, volume 4327 of *LNCS*, pages 123–140. Springer, 2006.

[11] D. Kalofonos and T. J. Norman. An investigation into team-based planning. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, pages 5590–5595, 2004.

[12] F. Meneguzzi and M. Luck. Composing high-level plans for declarative agent programming. In *Proceedings of the Fifth Workshop on Declarative Agent Languages*, pages 115–130, 2007.

[13] Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Proceedings of the First Workshop on Declarative Agent Languages and Technologies*, volume 2990 of *LNCS*, pages 135–154. Springer, 2003.

[14] S. Munroe, M. Luck, and M. d'Inverno. Motivation-based selection of negotiation partners. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1520–1521, 2004.

[15] Organization for the Advancement of Structured Information Standards. Introduction to UDDI:Important Features and Functional Concepts. Online, 2004. http://uddi.xml.org/files/uddi-tech-wp.pdf.

[16] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

[17] S. Sardina and L. Padgham. Goals in the context of BDI plan failure and planning. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 16–23, 2007.

[18] J. R. Searle. *Speech Acts : An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[19] M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[20] J. Thangarajah, J. Harland, D. Morley, and N. Yorke-Smith. Aborting tasks in BDI agents. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, 2007.

[21] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning*, pages 470–481. Morgan Kaufmann, 2002.

[22] J. F. Zhang, X. T. Nguyen, and R. Kowalczyk. Graph-based multi-agent replanning algorithm. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 793–800, 2007.