

Plans, Actions and Dialogues using Linear Logic

Lucas Dixon, Alan Smaill, Tracy Tsang

Preprint: 20 Feb 2009

Abstract

We describe how Intuitionistic Linear Logic can be used to provide a unified logical account for agents to find and execute plans. This account supports the modelling of agent interaction, including dialogue; allows agents to be robust to unexpected events and failures; and supports significant reuse of agent specifications. The framework has been implemented and several case studies have been considered. Further applications include human-computer interfaces as well as agent interaction in the semantic web.

1 Introduction

In recent years there has been a rapid growth of the distributed exchange and transformation of structured information in networked systems [44, 5, 45]. Components in these systems reason with the semantic information and communicate over the network. This naturally leads to the components being viewed as agents and, correspondingly, communication over the network can be thought of as a dialogue between agents. Agent dialogue is thus an increasingly important area of research that relates the semantic information with the agent's behaviour [37, 47]. Even before these applications, dialogue between agents was studied both for the long-term vision of robots that need to collaborate [35] and as a basis for cognitive models of human dialogues [9]. The close correspondence between language and planning, as investigated by Greenfield [18] and more recently Steedman [41], among others, suggests that planning can provide general purpose machinery to support agent interaction. Thus, providing a clear logical account for agent dialogue with an implementation is of significant interest and has wide reaching applications.

With these motivations in mind, this paper introduces a framework for specifying agents who reason by planning and interact through dialogue. The agents execute their plans and react to unexpected events and failures. In particular, we follow the philosophy that speech acts as well as other kinds of actions function in the same way as planning operators [9]. In this style, we provide a logical foundation, using a fragment of Linear Logic [17], for specifying agents. Specifically, a proof in Intuitionistic Linear logic (ILL) can be viewed as an agent's plan. Such a plan contains the agent's actions, including speech acts, as well as those of other agents. This provides a logical foundation for an agent based system which gives an account of why agents do what they do, and in particular, of why they say what they say.

This foundational theory for agents is pleasingly simple: the state of an agent is characterised by a Linear Logic sequent. The agent finds a proof of the sequent which corresponds to a plan. ILL has also been given a computational reading [1], which means that the plans found correspond systematically to executable programs in a linear functional programming language; we use this reading of ILL to characterise plan execution.

We also provide support for failure of plan execution: once a plan is found, it is executed until either a step fails or the plan is successfully completed. The result of execution is thus a modification

of the available resources; success corresponds to using up all resources and achieving all goals, and failure occurs when execution does not proceed as the agent planned it to. Unplanned resources can be produced and resources which were planned to be used up may not be. In such situations, the agent re-plans to try and find another solution given the available resources in the new state.

In summary, the proposed framework provides:

- A clear logical foundation for planning and the subsequent execution of the plan by an agent. Furthermore, the planning and execution mechanism is independent of the application domain.
- A model of dialogue which eliminates the need for explicit and fixed protocols, thus allowing an agent to interact with different agents in different ways via a more flexible planning approach.
- A treatment of imperfect understanding and failed actions. These have recently been highlighted by Foster et al. as an important area of further work for dialogue systems [15]. As well as being of practical importance, a treatment of unexpected events and failures which arise during the course of a dialogue is also a requirement for a cognitively plausible account of dialogue.
- Support for abstract plans in the sense that they do not need to include all of the steps which will be taken. We illustrate how resources representing an ‘imaginary’ state can be used to allow a lazy approach to planning, only filling out the details of the plan when necessary at execution time.
- A language for specifying agents which makes many features usable across domains, giving a practical level of reuse for agent specification. For instance, we show how to implement a generic notion of questioning and answering that reflects the corresponding speech acts.
- A model of agent interaction which is independent of the Gricean principle of cooperation. Following the work of Traum and Allan, who observe the problem that agents often need shared goals and plans in order to interact [46], agents in our framework are instead motivated to communicate by the believed effects of the actions, including speech acts. This provides a flexible approach to communication that lets the agent designer decide the level of detail that is used to represent other agents as well as the nature of interaction. Thus plan recognition and shared intentions are not needed for collaboration.

An implementation of our approach has been developed in a distributed architecture that allows the agents to be executed on different machines. We show how a distributed programming framework, Alice [2], in which the Linear Logic programming language Lolli [21] has been ported and extended, can provide a natural and declarative way to implement our approach. All case studies and corresponding code can found on-line.¹

In terms of the plans that the agents make, we remark that using Linear Logic terms instead of STRIPS style plans overcomes traditional limitations. Namely, new objects can be introduced and plans with conditional branching and sensing actions can be made. These observations have previously been made (e.g. [28, 11, 13]), but the work in this paper shows the practical use of these features of Linear Logic. We also make significant use of another feature of Linear Logic not available to planning formalisms: the need to remove all resources for a proof to be completed. We use this to model an agent’s expectations and obligations.

An interesting feature of our approach is that agents do not need to use the same internal representations of the world. While they can have different representations of beliefs, we do require that agents share a sufficient amount of communication primitives.

¹<http://dream.inf.ed.ac.uk/projects/dialoguell/>

1.1 Overview

In §2 we give the necessary background on Intuitionistic Linear Logic, our notation for it and its correspondence to planning. We present related work in §3 and then describe our framework for specifying agent goals, planning and executing plans in §4. We give details for representing an agent’s knowledge in §5. This covers basic synchronisation of speech acts, support for context-aware responses, and considers working with partial knowledge and the treatment of negation. In §6 we describe how our framework is implemented in the Linear Logic programming language Lolli. We provide case studies in two domains. The first domain considers agents buying and selling coffee. In this domain, we examine case studies for synchronisation of questions and answers in §7.1.1; we extend the study to consider interruptions, execution failure and subsequent replanning in §7.1.2; an example of working with incomplete knowledge is presented in §7.1.3; and use of conditional planning is shown in §7.1.4. The other domain, discussed in §7.2, is Power’s door problem [35]. In this case study we illustrate how planning can be performed in a lazy way by intentionally designed execution failure that leads to replanning. This avoids the need to make plans in full detail. With these case studies in mind, we evaluate the features and difficulties of our approach in §8. Finally, we conclude and describe avenues of further work in §9 and §10 respectively.

2 Background

2.1 Intuitionistic Linear Logic

Originally proposed by Girard [16], Linear Logic differs from traditional logic in that it is resource-sensitive. We work with an intuitionistic version of this logic: Intuitionistic Linear Logic (ILL), presented in sequent form. A sequent has the shape $\Gamma \vdash G$, where Γ is a list of ILL formulas, and G is an ILL formula. A proof in ILL describes how the resources in Γ can be consumed to produce those in G . Each antecedent corresponds to a resource which can be used exactly once, and so two copies of the same resource are treated distinctly, rather than as one. Linear Logic achieves this by not having the weakening and contraction rules. We consider ILL since proofs in this fragment can be given a direct computational reading [1]; in particular, there is a close correspondence to planning [28, 11].

A summary of the notation we use for ILL is as follows:

Linear Implication: $A \multimap B$ (read: *A lolli B*) indicates that the resource *A* is consumed and the resource *B* is produced as a result.

e.g. “`eating`: `hungry` \multimap `full`” indicates that the act of eating has the effect of removing a state of being hungry and generating a new state of being full. In a computational reading, the name “`eating`” refers to the action of eating and has the Linear Logic type “`hungry` \multimap `full`” which specifies the behaviour of the action. This shows the notation we use for relating planning actions to their logical specification. We omit the action’s name when it is not of interest.

Multiplicative Conjunction: $A \otimes B$ indicates that both resources *A* and *B* are present.

e.g. “`euro` \otimes `euro` \multimap `cake`” expresses that using up 2 euros can produce a cake.

Additive Disjunction: $A \oplus B$ indicates that either resource *A* or *B* is present, but not both.

e.g. “`lottery_ticket` \multimap `win` \oplus `lose`” means that a lottery ticket can be used to make one win or lose, but one cannot choose the outcome.

Additive Conjunction: $A \& B$ indicates that one can choose to have resource *A* or resource *B*, but the choice is exclusive.

e.g. “euro \multimap tea & coffee” expresses that a euro gives one the choice of either buying a tea or a coffee.

Bang: $!A$ indicates that one can get an arbitrary number of copies of the resource A .

e.g. having “ $!(\text{tea} \multimap \text{euro})$ ” as a resource can be used to express the ability to sell tea for a euro as many times as needed.

Note that the precedence is as follows in order of increasing strength: $\multimap, \oplus, \&, \otimes, !$. Thus the term $!A \oplus B \multimap C$ should be read as $((!A) \oplus B) \multimap C$.

Our formalism is based on Barber’s dual-context notation for our ILL sequents [4]. These sequents have two kinds of context: the first captures resources of which there are arbitrarily many and is called the non-linear or *intuitionistic* context; the second kind expresses ordinary resources which can be used up during planning and is called the *linear* context. To distinguish between resources in the linear context and those in the intuitionistic context, we use “**name** : **specification**” and “**name** !: **specification**” respectively.

We allow both the resource names and resources in ILL to be predicates and use first order schema with a curried notation for these. This giving rise to sequents such as:

```

buyfrom A X !: selling A X Y  $\otimes$  have Y  $\multimap$  have X,
sellingtea !: selling shop tea euro,
sellingcoffee !: selling shop coffee euro,
    p1 : have euro,
    p2 : have euro
 $\vdash$  have tea  $\otimes$  have coffee

```

where `buyfrom A X` is the name for a resource of type `selling A X Y \otimes have Y \multimap have X`. This expresses the ability to buy an object X from an agent A for the cost Y . The X , A and Y are schematic: they can be viewed as being universally quantified at the meta-level.

2.2 The Correspondence Between Linear Logic and Planning

In order to plan successfully, we must be able to keep track of effects of applying certain actions: what remains the same and what has changed. This is known as the frame problem and is treated in situation calculus by using explicit frame axioms [29]. Linear Logic provides an arguably simpler solution to this problem and fragments of it have been shown to have a close correspondence to planning [28, 11, 13]. In particular, a subset of intuitionistic fragment can be thought of as providing a logical foundation for STRIPS-style planning: full ILL is a more expressive formalism. Generally, the STRIPS style of planning eliminates the need for frame axioms by specifying actions as a collection of preconditions, facts to be added, and facts to be deleted.

ILL differs from traditional accounts of planning in that all the resources must be used up, where planning typically requires that the goal is achieved independently of other characteristics of the goal state. In this regard, the affine fragment of ILL has a more direct correspondence to planning and is decidable where ILL is undecidable [23]. However, rather than use the affine fragment, we add actions that use up selected resources not of interest to the final goal. This provides a flexible approach to ignoring characteristics of a goal state and maintains the ability to represent subgoals as resources that need to be removed because they cannot be ignored. This is an important feature of ILL and is used extensively in the case studies to capture the state of being expected to perform an action.

Another point of contrast between Linear Logic and STRIPS planners is that STRIPS assumes a fixed finite domain of objects in any given planning scenario. In contrast to this, Linear Logic allows

previously unseen resources to be generated. This feature allows agents to reason with new concepts that arise during dialogue. Linear logic also contains a choice connective, the additive disjunction, which can be used to make conditional plans. The Linear Logic additive conjunction can be used to represent a state in which an agent is offered a choice of resources, such as either buying coffee or buying tea. In summary, Linear Logic provides a simple but expressive logical account that extends planning. In the next section, we explain how ILL also supports plan execution.

2.3 Linear Logic and Plan Execution

ILL not only provides a framework for building plans, it also has a computational interpretation in terms of plan execution with an associated behavioural semantics. Here we follow the original proposal of Abramsky, as amended by Barber [1, 4].

Abramsky provided a systematic way to build programs from proofs in ILL. We use this view of ILL proofs to provide an interpretation of plans as programs and thus inherit a well defined model of execution. These programs are in a linear functional programming language, with an associated evaluation semantics.

Each inference rule in ILL is augmented to allow extraction of programs by suitable composition with the programs that correspond to the proofs of the premises of the rule. We write $\mathbf{x}_1 : \mathbf{H}_1, \mathbf{x}_2 : \mathbf{H}_2 \vdash \mathbf{G}$ for a sequent with resources $\mathbf{H}_1, \mathbf{H}_2$ on the left, indexed by $\mathbf{x}_1, \mathbf{x}_2$. The resource \mathbf{G} , on the right, is the conclusion.

Given a proof of a sequent:

$$\mathbf{x}_1 : \mathbf{H}_1, \mathbf{x}_2 : \mathbf{H}_2, \dots, \mathbf{x}_n : \mathbf{H}_n \vdash \mathbf{G}$$

we can automatically extract a program P with the following property:

If resources $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_n$ satisfy the specifications $\mathbf{H}_1, \mathbf{H}_2, \dots, \mathbf{H}_n$, then execution of the program based on these resources will terminate, yielding a resource that satisfies the specification \mathbf{G} .

For an example of how these programs are built up, consider the inference rule for \multimap on the left:

$$\frac{\mathbf{x}_1 : \mathbf{H}_1, \dots, \mathbf{x}_n : \mathbf{H}_n \vdash \mathbf{A} \quad \mathbf{B}, \mathbf{y}_1 : \mathbf{G}_1, \dots, \mathbf{y}_m : \mathbf{G}_m \vdash \mathbf{C}}{\mathbf{x}_1 : \mathbf{H}_1, \dots, \mathbf{x}_n : \mathbf{H}_n, (\mathbf{A} \multimap \mathbf{B}), \mathbf{y}_1 : \mathbf{G}_1, \dots, \mathbf{y}_m : \mathbf{G}_m \vdash \mathbf{C}}$$

In goal directed proof search, this allows us to solve the goal \mathbf{C} , given a resource $\mathbf{A} \multimap \mathbf{B}$ with a set of other resources, by partitioning the resources into two sets $(\mathbf{G}_i, \mathbf{H}_i)$, and finding proofs for the two sequents above the line.

Additional information tells us how to build a program for the resultant sequent, given proofs of the sequents above the line.

$$\frac{\mathbf{x}_1 : \mathbf{H}_1, \dots, \mathbf{x}_n : \mathbf{H}_n \vdash \mathbf{t} : \mathbf{A} \quad \mathbf{x} : \mathbf{B}, \mathbf{y}_1 : \mathbf{G}_1, \dots, \mathbf{y}_m : \mathbf{G}_m \vdash \mathbf{u} : \mathbf{C}}{\mathbf{x}_1 : \mathbf{H}_1, \dots, \mathbf{x}_n : \mathbf{H}_n, \mathbf{f} : \mathbf{A} \multimap \mathbf{B}, \mathbf{y}_1 : \mathbf{G}_1, \dots, \mathbf{y}_m : \mathbf{G}_m \vdash \mathbf{u}[\mathbf{f}(\mathbf{t})/\mathbf{x}] : \mathbf{C}}$$

The new annotations indicate that given a program \mathbf{t} that computes a value for \mathbf{A} , and a program \mathbf{u} that computes \mathbf{C} , we associate the resultant sequent with the program $\mathbf{u}[\mathbf{f}(\mathbf{t})/\mathbf{x}]$, constructed by substituting $\mathbf{f}(\mathbf{t})$ for \mathbf{x} in \mathbf{u} . Since program annotations are assigned to each inference rule and to the axioms, recursive application of these rules associates a program with every completed proof in ILL.

Execution of these programs follows the reduction relation given in [1], revised by [4].

3 Related Work

3.1 Relation to BDI Framework

The Belief, Desire, Intention (BDI) model is a popular way to view a reasoning agent’s ‘mental’ state [36]. For the agents in our system, the resources correspond to beliefs, the goal and any resources that cannot be ignored correspond to desires, and the plans generated using the available resources correspond to intentions. In our case studies, our agents were designed not to make inferences about the intentions of other agents. This simplified their design and showed that interpreting another agent’s intentions is not necessary for agents to interact.

Harland and Winikoff proposed an alternative view of agent-based systems where the foundations of the BDI agent model are re-cast into a framework based on classical Linear Logic [20]. Their aim is to provide an alternative logical account for BDI agent-based systems which allows them to be specified in a logic programming language. Our framework performs a similar function by allowing agent-based systems to be implemented in Lolli. The representations used are quite different from our framework. In particular, we allow each agent to be viewed as an ILL sequent. Our system has also been implemented and case studies considered.

3.2 Expectations and Obligations

Traum and Allen [46] proposed augmenting the traditional Belief, Desire and Intention framework with discourse obligations. The idea of obligations comes from the observation that people behave according to social norms. A discourse obligation is introduced, for example, when a question is asked, where an obligation for the other agent to respond is created. Like our use of expectation resources, Traum and Allen’s use of obligations provides an account for agent interaction when agents do not have joint intentions or reasons to adopt a shared goal. We observe that Traum and Allen’s concept of obligation also functions as an implicit form of protocol. A speech act causes an obligation to respond, thus it defines the way agents will interact in a similar way to a protocol.

While the use and effects of Traum and Allen’s obligations are very similar to our expectations, the underlying representation is quite different. They implement obligations outside of an individual agent as a set of rules which are directly connected with the actions that an agent is obliged to make. Furthermore, Traum and Allen’s obligations form a separate class of concept from beliefs, desires and intentions, where we model expectations simply as a new form of resource. Our use of expectations thus avoids having to extend the general framework, where obligations require additional machinery.

3.3 Dialogues as protocols

The relationship between speech acts and protocols has previously been represented explicitly by adjacency pairs [39]. Adjacency pairs are pairs of utterances made by different speakers, where the second is a response to the first, e.g. ‘question/answer’ and ‘offer/acceptance or rejection’. In our case studies we used expectations to describe aspects of the state of an agent that correspond to beliefs about the future. The use of expectation resources is more flexible than adjacency pairs as it allows no reply to be made in some cases and, similarly, supports a single reply to get out of many levels of nested conversation.

Increasingly, dialogues have been modelled as communication protocols. For instance, Fernández and Endriss give a formal account of various protocols as a basis for proving properties of dialogue systems [14]. However, using prescribed protocols also limits the ways in which agents interact. McGinnis, Robertson and Walton present a more flexible approach for agent communication by allowing them to synthesise interaction protocols [30]. Similarly, the work we present in this paper provides a formal

account of agents and the possible ways they can communicate. This allows fully formal proofs about dialogue systems and also removes the need for a predetermined communication protocol.

Another formalism for agent communication based on Linear Logic has been suggested by Künigas and Matskin [25]. They argued that distributed Linear Logic theorem proving could be applied for symbolic agent negotiation. Intuitionistic Linear Logic is used by Künigas and Matskin with a partial deduction mechanism to determine sub-tasks which require cooperation between agents. The main difference between our work and theirs is that we capture individual agents in our theorem proving process and the plan does not need to be shared between agents. This allows agents with different plans to cooperate and interact. Our case studies did not require the level of negotiation suggested by Künigas and Matskin, but their ideas could easily be implemented in our framework.

3.4 Plan Based Models of Dialogue

Models of dialogue based on planning, in which Austin’s speech acts [3] are viewed as planning operators, have been present since the end of the 1970s [9]. At about the same time, one of the first accounts of agents achieving goals by cooperating through dialogue was presented by Power [35]. In Power’s system, planning procedures are run individually by the agents to construct trees which indicate who is responsible for achieving each of the subgoals in a given plan. From this point on the agents both run according to a set of shared conversational procedures based on their allocated goals. These involve a control stack which keeps track of the currently running procedures. There are conversational procedures for both discussing the found plans and for executing them. More recent planning-based approaches to dialogue also use a fixed conversational procedure for resolving differences in agents’ beliefs [7, 8]. Like Power’s work, this approach provides protocols for dialogue that allow agents to collaborate and discuss differences in beliefs.

Our work provides an alternative foundation for collaborative dialogue. In our framework, we lift the restriction that agents have shared conversational procedures or fixed protocols for interaction: their internal beliefs about the effects of a speech act can vary. This allows our planning procedure to be used to guide conversation as well as to form plans. Our approach also provides a simple logical account of an agent. This comes from the relationship between planning and Linear Logic and can thus also be used for other plan-based approaches to agent dialogue.

In terms of Power’s case study, our main observation is that the representation of dialogue features makes a noticeable difference to the length of dialogue required to achieve a goal. As noted by Power, his mechanism for discussing plans makes even a simple task become tediously long. Our case studies suggest that many problems may be solved by a much lighter-weight approach to communication, one that does not require the use of rich shared structures such as plans.

Planning techniques have also recently been used for dialogue by Steedman and Petrick [42]. This is based on the Linear Dynamic Event Calculus (LDEC) which borrows aspects of Linear Logic, in combination with Dynamic Logic (to incorporate action sequences), and Kowalski and Sergot’s event calculus [40, 41]. Like the use of Linear Logic in our approach, this avoids the frame problem. Their work extends LDEC with planning mechanisms that support sensing and incomplete knowledge [33, 34]. To make planning more efficient they use specialised databases for different kinds of knowledge. An efficient implementation of our approach could also use such techniques. Their formalism does not include plan failure, although it has been used in combination with an execution monitoring system to provide failure handling in a similar way to our approach [24].

3.5 Information States and Linear Logic

Here we review briefly the connection between our Linear Logic formulation and the analysis of dialogue based on *information states* developed in the TRINDI project [26]. Roughly, the information state

encapsulates an agent’s beliefs about the state the dialogue is in, including private beliefs, the agent’s motivation, some record of the dialogue history, and obligations that the agent is committed to. Then various *dialogue moves* associated with actions trigger the update of information states of the agents involved. There is a set of *update rules* that set down how information states evolve for given dialogue moves. Generally there are several possible dialogue moves in a given context; some *update strategy* is needed to choose between the possible moves.

These components are related to our formulation as follows. A Linear Logic sequent corresponds to an information state, where:

- the motivational focus corresponds to the formula on the right hand side of the sequent; it may be a complex formula combining goals using conjunctions, disjunctions, or other connectives;
- the agent’s beliefs and commitments are represented as formulas in the context of the ILL sequent;
- dialogue moves correspond to our action operators;
- the inference rules of Linear Logic tell us how information states change when a given action is performed;
- finally, we use a planning algorithm to find a course of action that is believed to achieve the current goal. Different search strategies here correspond to the adoption of different update strategies.

In [26], the authors note that the “structural dialogue state” approach can easily subsume notions of dialogue state used on grammar-based formalisms. They also note that plan-based approaches, such as ours, are more amenable to flexible dialogue behaviour. They criticise plan-based approaches as more opaque, given “the lack of a well-founded semantics for plan-related operations” [26, p 325]. We believe that the Linear Logic approach provides just such a well-founded semantics, thus making the plan-based approach more attractive.

4 The Framework

The framework we present here provides a way to specify agents and gives a simple and uniform model for their reasoning and their execution of plans. An agent in our framework is represented by an ILL sequent of the form $\Gamma \vdash G$, where G is the agent’s goals; the linear resources in Γ characterise the agent’s state, while the intuitionistic resources in Γ characterise the agent’s actions.

Given such a specification, an agent’s reasoning process is simply ILL theorem proving. This corresponds to planning in the sense that the proof found is the agent’s plan. We use a proof search algorithm with a limited depth that finds the shortest plan first, such as iterative deepening. A naive unbounded search procedure would find plans with redundant steps or simply fail to terminate. Once a plan is found, the agent executes it until it achieves its goal or a step in the plan fails. Upon failure, the agent tries to find a new plan given the new state. This approach to failure in a non-deterministic environment is a form of execution monitoring and replanning [38].

Our approach is summarised in Figure 1 which gives an illustrative overview of how an agent interacts with the outside world by forming plans, executing them and interpreting sensed information.

The requirements of an environment for our agents to operate in are not difficult to meet: there must be a defined set of actions the agent can take and there must be some way for messages to be passed from the world to the agent. In terms of agent communication, we do not consider translation

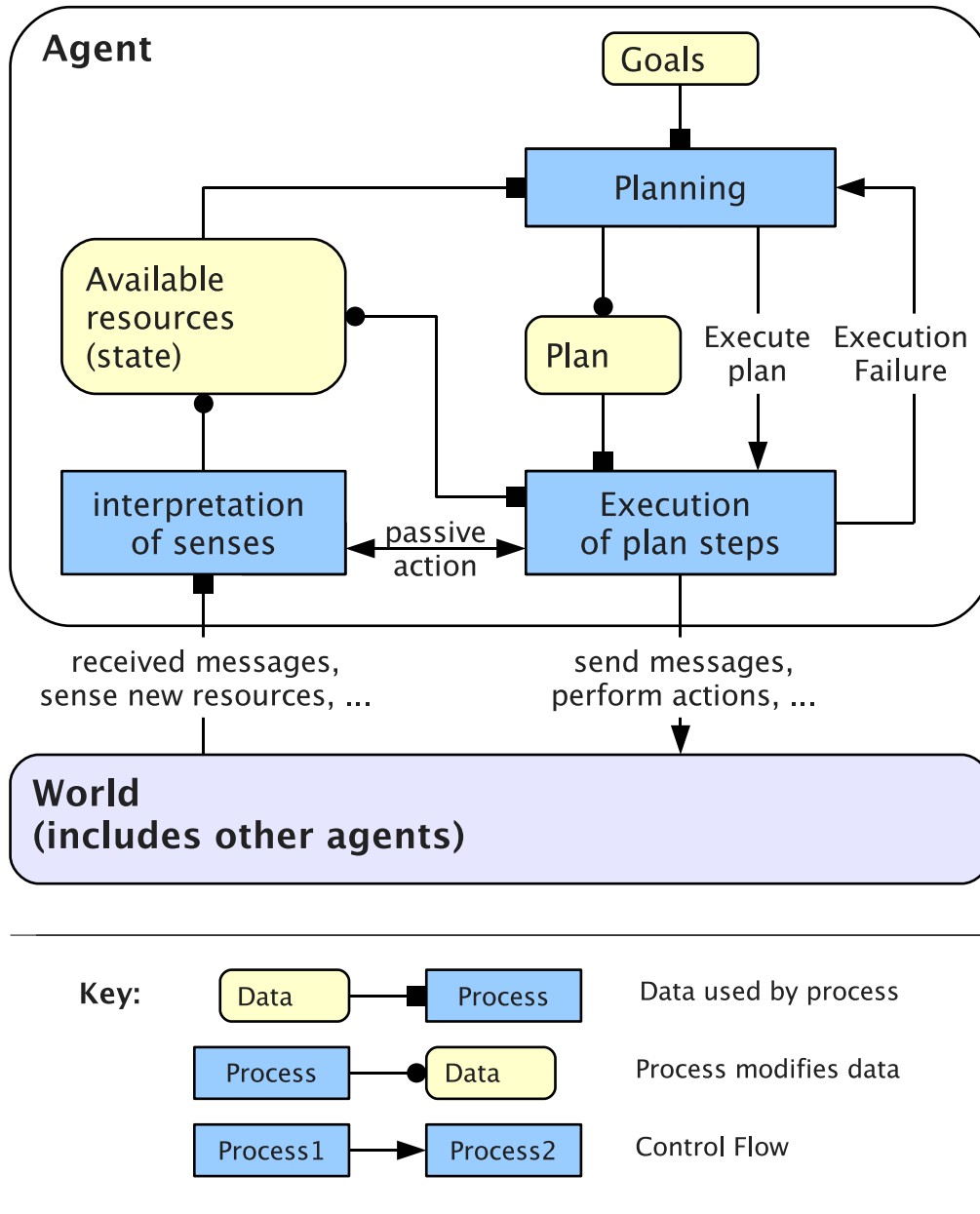


Figure 1: An overview of how an agent interacts with the world, describing control flow and showing the way data is used and updated. The agent performs actions in the world and observes information. Thus the world is both a source of data and a process.

of speech acts into natural language and vice versa as this is an orthogonal issue. Instead, we send predicates between the agents, and use canned text to make the output more easily readable for humans.

In the following subsections we describe in more detail how agents are specified, and how they interact with each other and the environment.

4.1 Agent Specifications

Agents are specified as a Linear Logic sequent in which the conclusion describes an agent's goals and the premises capture its current state and abilities. The state is made up of two kinds of resources:

External state resources refer to aspects of the world that are available to an agent. In our examples agents have direct access to knowledge of their possessions. This allows us to represent an agent having an object X as a resource of the form **have** X . Which kinds of resource fall into this category depends on the problem domain. External resources always correspond to the state of the world and thus such resources cannot be arbitrarily modified by the agent, though situating the agent in an appropriate environment may allow such modification.

Internal state resources correspond to the internal state of an agent and are specified by the agent designer. These do not necessarily have any correspondence to the state of the environment. How the internal state is modified by the perception is chosen by the agent designer. For instance, in our case studies (§7), when one agent listens to a question asked by another, it generates expectations which are modelled as internal resources.

For example, an agent which has two euros and is expecting to be given coffee would have resources that include **have** euro , **have** euro and **expect** coffee . The **have** euro resources could be modelled as part of the external state — under the assumption that the agent has direct knowledge of what it possesses. On the other hand, the resource **expect** coffee is part of the internal state as there is no guarantee that it corresponds to anything in the world.

The steps in an agent's plan, called *action operators*, modify both kinds of state resources.

Action operators are resources that describe an agent's belief about actions. Typically, they are of the form:

$$\text{ActionName } \text{Parameters} \ !: \ \text{Consumed} \multimap \text{Generated}$$

the parameters to the action (*Parameters*) allow us describe classes of actions. The resources used up are *Consumed* and those produced are *Generated*. The reason actions are usually intuitionistic is that actions themselves do not get consumed when they are performed.

For example, an agent can use the following resources (as action operators) to describe its beliefs about asking another agent, A , for an object X , and subsequently getting the object from them:

```
ask A (give X) !:
  has A X  $\multimap$  has A X  $\otimes$  expect A (togiveme X)
getfrom A X !:
  expect A (togiveme X)  $\otimes$  has A X  $\multimap$  have X
```

The `ask` action generates an expectation that another agent `A` will give the object `X` to the agent who has just asked for it. The action requires the resource `has A X` to be present but does not remove it because it occurs on both sides of the lolli. This limits asking for an object to cases when the asking agent knows the other agent has the object and is thus called a *precondition* resource. The `getfrom` action requires the agent to be expecting to be given `X` from `A`. This illustrates the way conversational context can be built and used to structure actions and dialogue: the agent must ask to be given `X` from `A` before the agent can actually be given `X`. The `getfrom` action operator expresses that the agent believes that executing the action will remove the expectation and result in having the object being asked for. Furthermore, `A` will no longer have the object.

All actions that an agent believes can happen are specified as action operators. To allow plans to be made which involve other agents' actions, the set of action operators includes the actions taken by other agents. We call such actions *passive* because they typically involve waiting and perceiving something in the world rather than causing it to happen. For example, the act of getting an answer to a question is passive and can be written as:

$$\text{getanswer } Q \ !: \text{ expect (reply } Q) \multimap \text{ answer } Q \ A$$

This says that if an agent is expecting a reply to a question `Q`, then the `getanswer` action will remove that expectation and produce a new answer resource.

The choice connective in Linear Logic allows agents to operate in a non-deterministic environment, such as the semantic web, where the outcome of actions is not known in advance, but the possible outcomes are. At the logical level, non-determinism corresponds to a resource of the form $A \oplus B$. When such a resource is the consequence of an action, either the resource `A` or the resource `B` will be produced (but not both).

For example, the action of getting an answer to a yes-or-no question could be written as:

$$\begin{aligned} \text{get_yn_answer } Q \ !: \\ \text{expect (reply } Q) \multimap \text{ answer } Q \ \text{yes} \oplus \text{ answer } Q \ \text{no} \end{aligned}$$

Plans with actions that produce non-deterministic resources have a branch for each resource that can be produced. These are called contingent plans. Such plans allow agents to work in a non-deterministic environment by planning for the different contingencies. A detailed description of plans is given in §6.3.1 and a case study which uses them is discussed in §7.1.

4.2 Agent-World Interaction and Plan Execution

In the sequel, we refer to the executable counterpart of ILL proofs as *plans*.

Once a proof of the agent's goal has been found, the agent starts to execute the plan. The process of execution is defined by the evaluation relation of the operational semantics of ILL [1, 4]. This view of evaluation gives the intuitive concept of plan execution without failure. Thus both the processes of planning and execution are captured by ILL and related to each other appropriately: if a plan has been found then its execution will terminate and produce the expected state.

As presented in §2.3, the found plan is an ILL proof term involving the available resources. The action operators can be defined as additional axioms that correspond to their actual effect, or they can be programmed directly in ILL. Execution is simply the evaluation relation applied to the plan and the resources that describe the agent's state. In this way, both the agent's internal process and the agent's interaction with the world are in fact characterised by ILL. We describe in operational terms the treatment of how failure is detected and dealt with.

Before each action is executed (that is, before the evaluation relation is applied to the current action and state), the external state is checked to ensure that it contains the resources specified by the action’s premises. If all needed resources are available then the action is executed and the internal state is transformed as specified by the action operator. The external state is assumed to be affected by the action’s effect on the environment. When some of the needed resources are not available, this indicates that the agent’s model of the world is incorrect or incomplete. Missing resources inform the agent that execution has failed. This happens before the action has been attempted and the agent simply starts planning again. The agent loops between planning and execution until a plan has been executed successfully.

Once a plan has been fully executed, failure is still possible. Failure after execution is detected by checking whether the goal has been achieved and that no additional resources are present. The latter form of failure bears an interesting contrast to traditional planning and execution where additional characteristics of the state are simply ignored. This is what gives ILL an adequate notion of context, as demonstrated in the case studies. Again, failure is dealt with by simply replanning.

5 Knowledge Representation

In the previous section we defined an agent as a sequent in ILL. We now give a more concrete view of agents and illustrate issues in their specification. This section describes the approach to knowledge representation which is used in our case studies (§7). This is just one of a wide variety of ways to specify agents in our framework. However, as our case studies illustrate, it highlights that our framework is sufficient to study interesting problems for multi-agent systems.

We first consider a basic class of external state resources for objects that an agent has possession of. We then consider the construction of a basic taxonomic hierarchy. Using these basic concepts, we introduce the use of expectation resources to represent the effects of common speech acts. We then consider an example in which these are extended to support the use of context during dialogue. We also present speech acts which use an attribute based representation to support agents with incomplete knowledge of the world.

5.1 Possession as an External Resource

To represent possession of an object X , an agent has an external state resource of the form `have X`. For example, `have coffee` is a resource available to an agent that has coffee. When an agent can sense that another agent, A , has an object Y , we use the external state resource `has A Y`.

Agents also have action operators to manipulate these resources. For example, an action operator for making coffee from `coffee_grains` and `hot_water` might introduce a new `have coffee` resource:

```
make_coffee !:
```

```
  have coffee_grains  $\otimes$  have hot_water  $\multimap$  have coffee
```

Such actions are typically external in the sense that they involve the agent manipulating objects in its environment. The actions that affect the environment are typically domain specific. Different environments allow different sets of possible actions.

5.2 A Taxonomic Hierarchy

Representing the resources available to an agent and their ontological relationships can be done in many ways and is the subject of much research [12]. The choice of ontological machinery is an

orthogonal issue to dialogue management; use of other ontology systems would be an interesting area of further work.

In this paper, we use a simple mechanism to define the taxonomic hierarchy of resources within an agent. In our approach, a predicate defines a class of objects. Instances of the class are represented as arguments of the predicate. For example, a class of drinks which includes tea and coffee can be represented using a predicate `drink` of arity one, where the argument can be either `tea` or `coffee`, giving e.g. `drink tea`. In a logic programming style, this allows unification with the pattern `drink X` to match anything which is a drink. This approach causes the terms to get larger as the complexity of the taxonomy increases. However, this approach is simple to use as it requires no extra or special purpose ontology machinery.

5.3 Common Speech Acts

The basic speech acts that we use are the asking and answering of questions. These provide the fundamental synchronisation for dialogue and are then refined to provide other kinds of speech acts such as offering alternatives and giving further details about the characteristics of an object.

5.3.1 Asking Questions and Getting Replies

Asking generates the expectation of some kind of response from another agent, while getting asked causes the agent to believe that another agent is expecting something from it. This allows us to slightly generalise the action operators for asking and getting-asked, from getting a specific answer to a question, to simply getting some kind of response:

```
ask A ... !: ...  $\multimap$  iexpectfrom A (reply Q)  $\otimes$  ...
getasked ... Q !: ...  $\multimap$  theyexpect A (reply Q)  $\otimes$  ...
```

where the ‘...’ stands for other parameters, extra consumed resources, and produced resources respectively. Specific kinds of questions typically have additional used and produced resources. The parameter `A` is the agent being asked, or the agent waiting for a reply respectively; and `Q` is the question.

Answers to questions depend on the question being asked, but the general schemas for answering and getting an answer are:

```
answer A ... !: ...  $\otimes$  theyexpect A (reply Q)  $\multimap$  ...
getanswer A ... !: ...  $\otimes$  iexpectfrom A (reply Q)  $\multimap$  ...
```

The generation of the expectation resources when a question is asked, and symmetric consumption of them in answering, have the effect of synchronising agent communication. This basic machinery is useful to many domains and was used in all of our case studies.

5.3.2 Context-Sensitive Speech Acts such as ‘yes.’

A feature of natural language dialogues is the use of the dialogue context to allow succinct answers to questions. For example, when asked if you would like cup of tea, you can answer by simply saying ‘yes’. Because asking a question generates a resource in the Linear Logic context, we can easily model this behaviour.

One example of a speech act that works on a compound context is the making of an alternative offer, to which the answer ‘yes’ or ‘no’ can then be given. Alternative offers can happen when an agent has an expectation to give another agent an object. Offering an agent `A` an alternative object `X` instead of the object `Y`, which it was asked for, is also an instance of the `ask` action:

```

ask A (alternative_offer X) !:
  have X  $\otimes$  theyexpect A (tobegiven Y)
   $\multimap$  have X  $\otimes$  theyexpect A (tobegiven Y)
     $\otimes$  iexpectfrom A (reply (alternative_offer X))

```

For an agent to respond to such an offer, a simple ‘yes’ will suffice. The condition under which an agent can just say ‘yes’ is precisely the context in which such an offer has been made. Thus the action to get a ‘yes’ answer can be specified as:

```

getanswer A yes !:
  iexpectfrom A (reply (alternative_offer X))
   $\otimes$  theyexpect A (tobegiven Y)
   $\multimap$  theyexpect A (tobegiven X)

```

On the other hand, answering ‘no’ can be specified as consuming the expectation for a reply without generating any new resources:

```

getanswer A no !: iexpectfrom A (reply (offer X))  $\multimap$  1

```

where 1 is the tensor unit that comes from Girard’s Linear Logic and expresses that the action does not produce any new resources.

Management of context highlights an important issue for agent design: how much information is needed for agents to interact while avoiding infinite conversational loops. In the above scenario, it is easy to see that two agents can have an infinite conversation of the form:

```

A1: I would like some tea please.
A2: would you like coffee instead?
A1: no.
A2: would you like coffee instead?
A1: no.
...

```

Luckily this is can easily be fixed by adding a new resource to the context that changes when an offer is made and in particular when an alternative offer is rejected. The natural concept for such an implementation would be to check for the non-existence of having made an offer previously. The way we implement this in ILL is through the introduction of a resource which represents having not made an offer. Another solution is to make use of contingent planning and change the getanswer action to produce an additive disjunction (\oplus) of the yes and no actions. This highlights the role and importance of agent knowledge engineering in our framework.

5.3.3 Partial Knowledge as Attributes

Partial knowledge about an object in the world can be represented using attribute predicates. For example, to represent a coffee that is white and has sugar, we could use the resources `know(attr coffee colour white)` and `know(attr coffee sugar has_sugar)` respectively.

When an agent would like to ask about properties of an object, it can then use an instance of the ask question:

```

ask A (qattr X Att) !:
  (1  $\multimap$  iexpectfrom A (reply (qattr X Att)))

```

where X is the object and Att is the attribute being asked about.

An action operator to answer such a question is:

```
answer A (attr X Att Y) !:
  know (attr X Att Y)  $\otimes$  theyexpect A (reply (qattr X Att))
   $\multimap$  know (attr X Att Y)  $\otimes$  knows A (attr X Att Y)
```

This specifies an honest agent that must know the value of the attribute to give an answer - it has a resource of the form `know (attr X Att Y)`. The result is that a new resource is created for the belief that the other agent now knows the value of the attribute.

5.4 Ignoring

In our case studies, beliefs about possessions and knowledge were not important to the solution of a plan unless they are part of the goal, but other agents' expectations should not be ignored. This was important as the Linear Logic proof process works on the basis of using up all the leftover resources. We introduced ignore actions to give flexibility over the admission of proofs with certain kinds of extra resources:

```
ignore_has    !: has A X  $\multimap$  1
ignore_have   !: have X  $\multimap$  1
ignore_know   !: know X  $\multimap$  1
ignore_knows !: knows A X  $\multimap$  1
```

This allows a fine grained control over what can be ignored. Different agents can ignore different things depending on the agent designer's desires.

6 Implementation

The presentation given in the previous section assumes a Linear Logic theorem prover to perform planning and some basic machinery to interact with the world. In this section, we describe our implementation, which uses a Linear Logic programming language extended with two primitives for agent communication. Our implementation is expressed in the purely declarative fragment of Lolli.²

6.1 Lolli: Logic Programming in ILL

Lolli is a Linear Logic programming language designed by Hodas and Miller [21], based on a fragment of Intuitionistic Linear Logic. In this language, resources are predicates and, as in Prolog, goals are proved by depth first proof search. Lolli deviates from Prolog not only in the linear nature of resources, which are used like Prolog facts, but also in that linear implications can occur in goals.

Lolli allows queries such as:

```
?- (a,a)  $\multimap$  a.
no
```

```
?- (a,b)  $\multimap$  (b,a).
solved.
```

²The interested reader can consult the Lolli source code for execution and planning as given in Appendix B

where ‘?- ’ is the Lolli query prompt, ‘-o’ is the \multimap connective, and ‘.’ is used to end the query in the same way as in Prolog. The ‘,’ in Lolli denotes the \otimes connective. Lolli cannot prove ‘ $a \otimes a \multimap a$ ’ but can prove $a \otimes b \multimap b \otimes a$. Lolli attempts to solve the goal on the right hand side of the \multimap using the resources on the left hand side. The first query fails as there is a left-over resource ‘a’ which came from the left-hand side. The second query succeeds since the resources on the right correspond exactly to those on the left. Lolli provides the underlying theorem proving mechanism of our implementation.

6.2 Extending Lolli in Alice for Agent Communication

Alice is a implementation of the Standard ML programming language with support for concurrent, distributed, and constraint programming [2]. By porting the logic programming language Lolli to Alice, we were then able to extend it with two new primitives which provide the basis for agent communication:

- “put_chan C X” puts the value X onto channel C.
- “get_chan C X” gets a value from the channel C and then unifies it with X.

Each agent has a channel that it listens to and a channel that it send communications to. By connecting the channels appropriately, agents can participate in a dialogue.

6.3 Plans in Lolli

The main difference between the abstract presentation of our framework in §4 and that provided by Lolli concerns the representation of proofs. To get the desired representation of plans from Lolli, we provide an interpreted representation of plans and write the planner as a Lolli program. The planner we have implemented is based on an iterative deepening search with a specified bound. This meets the assumptions of our framework (§4): by finding shortest plans first, it avoids finding plans with redundant steps.

6.3.1 Plans

Plans are of the following form:

```
plan ::= id
      | then(action, plan)
      | par(plan, plan)
      | case(action, resource, plan, resource, plan)
```

Briefly, the operational semantics is as follows:

- id is the identity plan that does nothing and tries to ignore all ignorable resources;
- then(a, p) is a plan that performs the action a then performs the actions in the plan p;
- par(p1, p2) is a plan that performs the steps in p1 in parallel with those in p2³;

³In our implementation, we serialise the execution to perform p1 first, then p2. True parallel execution, as allowed by the semantics, is left as further work.

- `case(a, r1, p1, r2, p2)` is the plan that performs the action `a` which will produce either the resource `r1`, in which case plan `p1` will then be performed, or `a` will produce the resource `r2` and continue to perform the plan `p2`. To make case statements easier to read we write them as:

`case a of r1 ⇒ p1 | r2 ⇒ p2`

An example of conditional planning is given in §7.1.4.

6.3.2 Action Operators

The notation we introduced earlier for action operators was:

`ActionName Parameters !: Consumed → Generated`

This is translated into the intuitionistic Lolli resource:

```
actionop (ActionName Parameters)
  IPreconditions, IUsed, IProduced,
  EPreconditions, EUsed, EProduced.
```

where, as in traditional STRIPS planning, the common resources in *Consumed* and *Generated* are **Preconditions**. The rest of the consumed resources are in **Used** and the rest of the generated resources are in **Produced**. We separate, using the argument positions, the internal resources (those prefixed by **I**) from the external ones (those prefixed by **E**). This can also be done through a distinction in the ontology of resources.

To illustrate an action operator in Lolli, we show `answer A (attr X Att Y)` which describes the act of answering a question regarding an attribute of an object. The Linear Logic characterisation of this is:

```
answer A (attr X Att Y) !:
  iexpectfrom A X
  ⊗ know (attr X Att Y)
  ⊗ theyexpect A (reply (qattr X Att))
  → iexpectfrom A X
  ⊗ know (attr X Att Y)
  ⊗ knows A (attr X Att Y)
```

which we implemented in Lolli as:

```
actionop (answer A (attr X Att Y))
  (know (attr X Att Y), iexpectfrom A X)
  (theyexpect A (reply (qattr X Att)))
  (knows A (attr X Att Y))
  true true true.
```

where `true` In Lolli corresponds to the tensor unit `1` and indicated that no resource is used or produced. By stating that all external resources are `true`, the agent does not believe that the answer action will be affected by anything it can directly sense.

6.4 Execution in the Environment

Each action operator has a corresponding execution operator which performs the action in the environment. The effect of the action may not be the same as the agent's specification. In particular, this happens when the agent has incomplete or incorrect knowledge about the environment or consequences of an action.

To model a simple environment in ILL, as needed for our case studies, we specify execution operators as Lolli rules of the following form:

```
executeop ActionName Produced :-  
    Used.
```

where the `'-'` operator in Lolli is equivalent to the \rightarrow operator but with the argument-order swapped i.e. `A :- B` is equivalent to `B \rightarrow A`. The term `ActionName` is the name of the action operator being executed.

For example, a very simple environment which asks the user to input the effect for every action can be defined as follows:

```
executeop ActionName Produced :-  
    write ActionName,  
    nl, write "Enter resources consumed ('true' for none):" ,  
    read Used, Used,  
    nl, write "Enter resources produced ('true' for none):" ,  
    nl, read Produced.
```

Actions that allow agents to directly communicate with each other can be implemented using the `get_chan C X` function, where `C` is the channel name. The term that is received instantiates the variable `X` and corresponds to the result of the sensing action. Similarly, the `put_chan C X` can be used for to send a sensed observation to an agent. Using these functions allows agents to be run on different machines asynchronously. The interested reader can consult the implementation of the case studies for more details of using the channel operations and providing an environment.

7 Case Studies

To illustrate our framework and evaluate it, we implemented case studies in two domains:

Buying and selling coffee: one agent is trying to buy coffee while the other is selling it. We use this domain to consider basic synchronisation of communication, interruptions and unexpected actions in dialogue that cause execution failure, working with incomplete knowledge, basic negotiation by offering an alternative, and conditional planning.

Power's Door Problem: we have re-implemented the scenario described by Power [35] in our framework and used this as a point of comparison with his work. This case study also illustrates how planning can be delayed until execution time, thus allowing a lazy and incremental style of planning.

7.1 The Domain of Buying and Selling Coffee

Considering the scenario of dialogues in a coffee shop was motivated by its possible application to agents in a semantic web, such as on-line shopping assistants. Our first study only involves a buying agent requesting and then receiving a cup of coffee from a selling agent. From this simple synchronisation, we build examples to illustrate how more interesting features of dialogue can be implemented.

7.1.1 Basic Synchronisation: Request/Give

As the basis for our other examples in this domain, we implemented agents to request and subsequently get the requested object. This involves two agents which we will call `requester` and `giver`.

The act of requesting an object is an instance of the general asking action and correspondingly has the following action operators associated with it:

```
ask A (giveme X) !:  
  has A X  $\rightarrow$  (has A X)  $\otimes$  iexpectfrom A (togiveme X)  
getasked A (givethem X) !:  
  1  $\rightarrow$  theyexpect A (tobegiven X)
```

By asking to be given an object, an expectation to be given the object is generated. Conversely, getting asked to give an object generates an expectation to give the receiver the object. Notice that this example illustrates how our framework allows a representation of actions that is not symmetric with respect to the agent's perspective. In the above example, the agent can plan to ask for an object only when the other agent has the object, but the agent can plan to get asked for an object even if it is not known that the giver has the object. This feature illustrates the role of the agent designer as well as the flexibility she has in specifying agents.

An action to use up the expectation resource, which is produced from such a question, is to give the object in question to the other agent:

```
giveto A X !:  
  have X  $\otimes$  theyexpect A (tobegiven X)  $\rightarrow$  has A X  
getfrom A X !:  
  has A X  $\otimes$  iexpectfrom A (togiveme X)  $\rightarrow$  have X
```

These specify that an agent can give away an object if the agent they are giving it to is expecting to be given the object.

In this example, `giver` has the goal `has requester coffee`. The goal of `requester` is the same as `giver`, modulo the perspective of the agent: `have coffee`. Both agents have the same specification in terms of action operators, but the agent `giver` has the additional resource `have coffee`. The result of this scenario is that `requester` forms the plan:

- ask giver (giveme coffee)
- getfrom giver coffee

and `giver` forms the plan:

- getasked requester (givethem coffee)
- giveto requester coffee

Execution of this plan succeeds trivially, giving the following transcript:

```
requester : sayto giver (giveme coffee)  
giver : giveto requester coffee
```

The result is that the agents achieve their goals and live happily ever after.

The `has` resource is external and thus corresponds to a sensed fact. We remark that the requester agent was not required to have any explicit beliefs about the other agent. Instead, agents in our framework model the behaviour of other agents implicitly within the action operators.

7.1.2 Interruptions, Execution Failure and Re-Planning

Handling unexpected events during dialogue is an important part of designing a robust framework for interoperable agents. When the unexpected event is a speech act, such as being asked a question, we want to design an agent that gives appropriate responses. Typically, this involves nested conversations.

Building upon our previous example, we transform the agents into *customer* and *seller* agents. The seller has the goal of having one euro for the drink of coffee. Thus, unlike the previous example, the agents now have different goals. In this example, we do not modify the customer agent; the customer does not know that he has to pay for coffee. Initially, the customer and seller find the following plans:

customer plan:

- ask seller (togiveme coffee)
- getfrom seller coffee

seller plan:

- getasked customer (togivethem coffee)
- ask customer (giveme euro)
- getfrom customer euro
- giveto customer coffee

However, the plans diverge at the second step which leads to the seller asking the customer to pay when the customer is expecting to be given the coffee. Thus the effect of the customer's `getfrom` action differs from his specification of the action. In particular, the customer finds himself with an extra expectation to answer the payment request from the seller. If the customer has a euro, he then finds the new plan:

- giveto seller euro
- getfrom seller coffee

If the customer does not have a euro, he fails to find a plan. When the customer has a euro, after finding the new plan he continues to execute it which then leads to successfully buying coffee.

This simple example illustrates how agents with different goals, and different expected interaction protocols, can still collaborate. Furthermore, the agents can achieve their goals without any discussion or analysis of each others goals and plans.

7.1.3 Incomplete Knowledge: Sugar?

Often additional information about objects is required in order for an agent to perform certain actions. In our framework, we model any additional information about an object in terms of attributes, as introduced in §5.3.3. We use another extension of the coffee example to illustrate this. We modify the seller so that she must make coffee before selling it, and the process of making coffee requires knowing if sugar should be added. Formally, we model this using the attribute `sugar`, which is used by the action operator for making coffee:

make coffee S !:

- know (attr sugar coffee S)
- \rightarrow have coffee \otimes know (attr sugar coffee S)

Given this action operator, the seller's plan becomes:

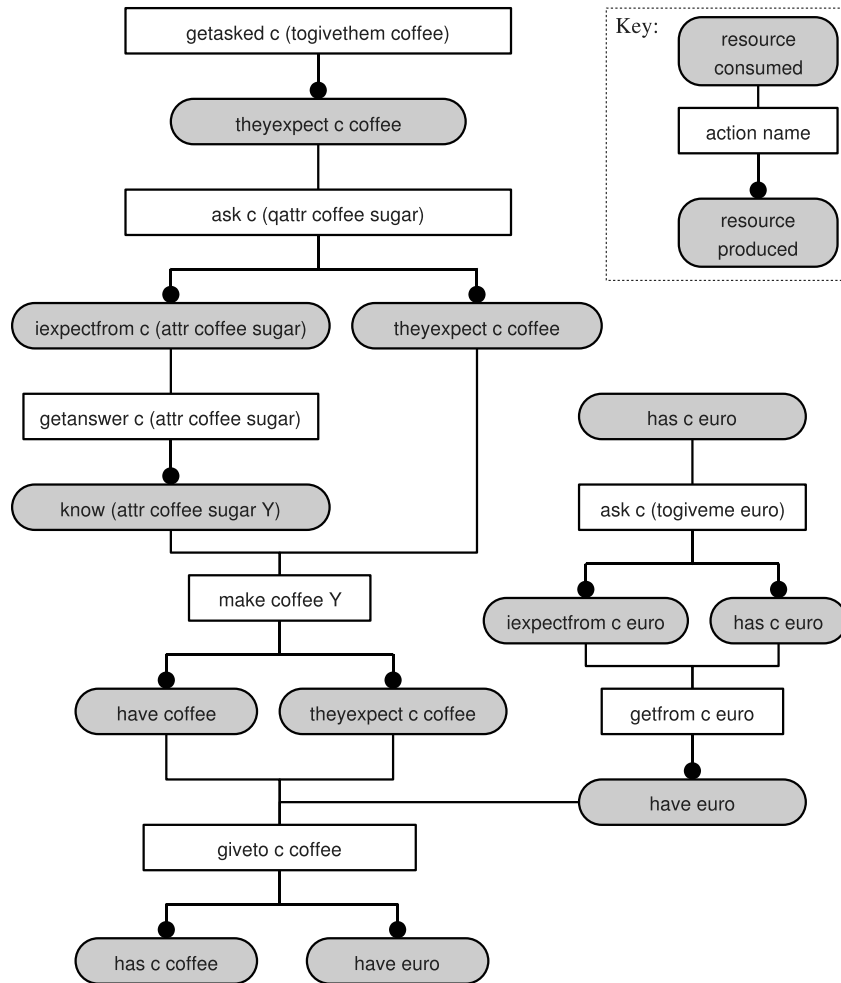


Figure 2: The seller's plan to sell coffee to the customer. The customer's name is abbreviated to *c* for brevity. Observe that, according to the theory of ILL, asking for a euro and then getting it, can be done in parallel to the rest of the plan.

seller plan:

- getasked customer (togivethem coffee)
- ask customer (giveme euro)
- getfrom customer euro
- ask customer (qattr coffee sugar)
- getanswer customer (attr coffee sugar S)
- make coffee S
- giveto customer coffee

The details of this plan, showing the resources produced and consumed, are illustrated in Figure 2. This shows that the above style of plan presentation is in fact a linearisation of the found (parallel) plan.

To enable the buyer agent to interact with this more complex seller, we need only give the buyer the generic action operators for talking about attributes in addition to the knowledge of how the buyer likes his coffee. Interestingly, we do not need to provide any further information concerning the protocol of interaction with the seller. The buyer's plan will fail but at execution time the buyer will

construct a new plan that involves answering the seller’s question(s).

7.1.4 Conditional Plans

Action operators can produce non-deterministic effects and agents can make plans for each contingency. For example, the act of getting asked for a tea or coffee could be specified as:

```
getasked A (request tea_or_coffee)
  !: 1  $\rightarrow$  (theyexpect A coffee  $\oplus$  theyexpect A tea)
```

This kind of non-deterministic specification of actions allows an agent to anticipate the different possible responses and make a conditional plan.

In particular, using this action will produce a plan-step of the form:

```
case getasked A (request tea_or_coffee) of
  theyexpect A coffee  $\Rightarrow$  CoffeePlan,
| theyexpect A tea  $\Rightarrow$  TeaPlan
```

where `CoffeePlan` is the plan for the case when executing the `getasked` actions produces the resource `theyexpect A coffee`, and `TeaPlan` is used in the other case when the resource `theyexpect A tea` is produced.

To extend our example to use conditional planning, the seller was modified to use the non-deterministic `getasked` action operator. The seller then makes and sells the corresponding drink. The customer does not need to be changed and behaves in the same way as previously.

7.1.5 Scaling-up by Combining Dialogue Features

To examine the consequences of mixing the features, and the ease with which they scale up to a larger example, we combined all of them into a single larger case study. There are two changes involved in this scaling up. The first is to stop the seller from making ‘rude’ plans in which she asks for payment before the customer has asked for a drink. A simple solution to this is to add a condition that restricts the seller to only asking for payment after the customer has requested a drink. The second modification is to the ontology used by the agents. The seller needs to distinguish between the objects it is selling and the money used to pay for them. Without this distinction, a devious buyer can trick our seller by purchasing `euro \otimes euro`, for the price of only `euro`. Clearly, this is bad business for our seller. However, it is easily fixed by making the appropriate ontological distinction: the seller only sells drinks for a euro.

7.2 Power’s Domain: Agent Collaboration

By re-implementing the case study presented by Power [35], we illustrate how agents in our framework are able to collaborate and overcome erroneous beliefs. Another feature of our solution to this domain is the use of an ‘optimistic’ action operator to represent a state at which the agent will perform further planning. This allows a lazy and incremental style of problem solving.

Power’s domain consists of two agents, John and Mary who share the goal of wanting John to get to the same place as Mary. There are two locations, `out` and `in`, and there is a door which can be `open` or `closed` between them. Mary is `in` but she is blind and thus unable to perceive the state of the door or the location of John. On the other hand, John can perceive the state of the world, but starts off `out` and lacks the knowledge that moving will change his location.⁴

⁴Power also includes a bolt on the door and limits John to being unable to push the door open. However, while these are easy to add to our domain, they play no role in this case study and thus do not discuss them here.

The problem is to specify the agents so that John and Mary can communicate and solve the problem of getting John *in*. The solution in [35] used a fixed protocol for how the agents communicate and in particular for the way they collaboratively work on a plan. This made use of a shared language of plans. The main part of the dialogue was involved in forming this shared plan. Essentially, this results in Mary asking John questions about the state of the world, opening the door for him, and instructing him to move.

The spirit of our framework is to make protocols implicit and flexible and let failed plans be fixed at execution time by replanning. An advantage of this is that dialogue can commence before a realistic plan has been found. By avoiding a complex procedure to talk about plans, we achieve a significant reduction in the number of dialogue steps, from 72 to 10.

7.2.1 Specifying the Agents

We gave Mary the goal of being helpful and John the goal of being *in*. In Power’s implementation both agents had to have the same goal. This was required in Power’s setting for the agents to collaborate. In our setting this is not necessary: by making Mary want to be helpful, she waits for someone to ask for help and then collaborates with them to solve their problem. The expectations of other agents provide sufficient incentive for collaboration. As in Power’s study, the production of natural language is not the purpose of the study, and thus canned text is used when desired.

The ‘physical’ actions available to the agents are the same as in Power’s account. John can move and open the door, but he has incorrect beliefs about the effects of moving. Mary has the same abilities and although she is blind, she knows the effect of opening doors and moving. To allow the agents to ask questions about the state of the world, we re-used the ability to ask about attributes which was initially developed in the coffee buying and selling domain.

7.2.2 The Resulting Dialogue

As in Power’s example, the setup results in John being unable to find a plan. Our solution starts by using an instance of the ask and answer actions to allow John to ask for help and then get instructions from Mary. Our solution combines this with an ‘overly optimistic’ action operator which represents the belief that following an instruction will solve the goal. The plan is overly optimistic because during execution the agent will typically be given several instructions and asked questions and so on. The action operator to get an instruction is really a placeholder for reacting to the advice given and performing further planning in the new situation. This allows us to specify agents that perform planning in a lazy and incremental fashion.

The resulting dialogue transcript, which shows the actions taken, is given in Figure 3. First, John asks for help from Mary. He believes that this will result in Mary giving him an instruction to follow that will achieve his goal. Instead, Mary asks a question. This causes John’s plan to fail; he re-plans, answers the questions and finally gets an instruction from Mary. Because Mary is looking for the shortest plan first, she decided to open the door for John and then tell him to move. Telling John to open the door would take more planning steps. However, this plan is found if we make Mary unable to open the door. Thus, although John’s plan was overly optimistic, and failed initially, the idea of the initial plan still worked: he asked for help and got it, and after various additional steps and the formation of several new plans, he achieved his goal. Further details of this case study can be found in Appendix A

- 1) John says: How do I achieve know (attr john loc in)?
- 2) Mary says: what is the loc of john?
- 3) John says: The loc of john is out
- 4) Mary says: what is the pos of door
- 5) John says: The pos of door is shut
- 6) ** Mary's action: pushes the door open
- 7) Mary says: OK, do movefrom loc out
- 8) ** John's action going to follow the instruction and
do movefrom loc out
- 9) John says: Ok, I've done that.
- 10) John says: My goal's been achieved! thankyou!

Figure 3: The dialogue between John and Mary, including the actions taken.

8 Evaluation

The general hypothesis being tested is the suitability of our framework for specifying agents and building systems involving them. Inspired by the questions Power asks of his dialogue system [35], and as a result of observations from our case studies, we have devised a set of topics for evaluating the planning and execution part of a dialogue system. In the following subsections we introduce the topics and summarise the issues and characteristics of our approach.

Representation of Goals and States

Agent behaviour is typically parametrised by having some initial internal state which includes a particular representation of its goals. During an agent's execution its internal state changes. The representation of goals and internal state is a key characteristic of an agent framework. It defines the ease with which agents can be specified as well as how easy it is to reason about an agent's behaviour.

In our system, the state and goals of an agent are represented as an ILL sequent. The state is described by the antecedent resources and the goals are the conclusions. This provides a simple way to view agents in ILL and provides a logical view which is amenable to formal reasoning. Using Linear Logic avoids frame axioms, resulting in what we believe is a simpler logical account than formalisms based on situation calculi. In particular, temporal constraints on an action come from its preconditions and no explicit notion of situation is needed.

Our formalism of an agent's state includes its abilities and beliefs. Within this representation, we have expressed having and giving objects, asking questions, context dependent answers, partial knowledge, and getting, following and giving instructions. These characteristics made up two quite different domains in which we felt that the specification of agents followed from our intuitions about expectations in human dialogue. We believe that this approach can also be used for formalising argumentation as well as other kinds of dialogue.

Reuse of Specification

Reuse is desirable as it simplifies developing new agents. In the coffee selling domain we were able to reuse all the code of earlier agents in later ones. Between the coffee selling domain and Power's door problem we were able to reuse the action operators for asking and answering questions as well as the partial knowledge representation. Given the difference between the two domains we consider this to be a high level of reuse.

Internal Model of Other Agents

How agents represent each other defines the way they respond to each other. For example, it defines whether agents are able to lie to each other. This raises questions about how an agent represents another agent's representation of itself, and so on.

In our framework, agents are not required to have an explicit model of other agents, but can rather model the actions which they believe others to be capable of. This leaves the level of detail in the representation of other agents up to the agent designer. Agents do not need to share plans or intentions, allowing our framework to be used for situations where agents have conflicting goals and plans as well as different internal models of the world. In our examples, the model of other agents is as resources representing their expectations. Our case studies suggest that the model of other agents does not need to be complex in order for quite sophisticated interactions to take place between them.

Maintenance of Context

Throughout a dialogue it is important to maintain contextual information so that later actions can depend on the results of earlier ones. The maintenance of context improves the flexibility of possible interactions, for example by allowing an agent to realise when it would otherwise start repeating its behaviour. However, it increases the amount of information that needs to be stored. The representation of context also defines the ease with which an agent can use the dialogue context.

Rather than using an explicit representation containing the dialogue history, we represent only the relevant information by using resources which characterise state within an agent. This decreases the amount of information the agent must store and provides a cleaner logical account of a dialogue's meaningful context. It is the agent designer who defines when her agent should add resources to, or remove them from, the context.

Ease of Specification

Practical implementation of an agent based system requires specifying the agents involved. The ease of this process is often best understood by the difficulties in specification and the guiding principles for specifying agents. The design for our framework originated from introspection on the cognitive process of dialogue planning and in particular on expectations as building blocks for dialogue plans. Linear Logic can provide a close correspondence with the effects of actions in the real world. Designing agents was largely an introspective process and getting the initial idea for the resources was easy. However, working in Lolli was sometimes problematic as debugging facilities are somewhat primitive. A typed system with an explicit specification of an agent's ontology would help significantly.

Another issue is the inability of an agent to identify the non-existence of a resource. For example, it can be useful to identify when another agent is not expecting an answer. Although this can be done by creating special negative resources, it requires the agent designer to ensure that it is not possible for an agent to have both a normal resource and the negative version of it. We believe this burden could be lifted to the level of the logical framework and suggest it as further work.

Plan Selection and Efficiency

In plan based systems of dialogue, a distinction can be made between the space of plans and the way a particular plan is selected. Typically, selecting a plan involves search. Our representation of plans is very rich and thus the space of plans is also very large. This introduces issues of efficiency.

In terms of search, we use an algorithm that finds the shortest plans first. This is motivated by wanting to avoid plans with redundant steps. However, we did not otherwise provide any bias to prefer

certain plans over others. Providing different search algorithms for ILL, such as best first, would allow some parametrisation of preferred plans at the expense of slightly complicating the nature of finding plans. Perhaps a more serious issue in our framework is that agents tend to be overly optimistic: agents find some action that another agent can take that will accomplish their goal. During execution, such plans typically fail and the agent replans. If this is considered problematic by the agent designer, it can be dealt with by providing conditional plans, or by generalising a collection of possible responses using a resource that intentionally causes replanning to occur at execution time.

As mentioned earlier, ILL is in general undecidable. Although a significant limitation of our system is the inefficiency of the naive planner's implementation, theorem proving techniques that remove symmetries in the search space can easily be implemented. For instance, whenever two actions are independent of each other, the order in which they are done is irrelevant and produces a symmetry in the search space. More efficient proof techniques have been considered by several authors [19, 10, 6, 27]. Our implementation of a naive depth bounded planner, which finds shortest plans first, finds the plans for our case studies within a number of seconds. With some of the aforementioned improvements, we believe our approach would also be effective for significantly larger examples.

Turn taking

With multiple agents in a dialogue, what is the mechanism for deciding who says what when? From an agent's perspective, how does it know when to wait for an answer and when to perform an action? Turn taking affects the flexibility of agent interaction, defines the possible interaction deadlocks and loops, and also influences the ability to reason about multi-agent systems.

In our framework, agents make plans which include the actions and speech acts of others. Since it is only possible to remove expectations for a response by responding appropriately, the protocol for turn taking is implicit in the agent's knowledge.

In terms of flexibility, this approach has the advantage of eliminating the need for an explicit representation of the interaction protocol. An agent can interact with different agents in different ways. It can keep track of which expectations correspond to which agents by tagging the expectation resources with the corresponding agent's name. Deadlocks and looping behaviour can happen, but one can consider the possible interactions and easily limit these problems. For instance, a timeout resource can be generated to handle deadlocks, and extra resources can be used to avoid looping.

Unexpected Events

Agents typically have an imperfect model of the environment, including other agents. This leads to unexpected events during execution. It is important to provide agents with some means to handle such eventualities.

We allow failures to be handled without having to pre-specify where failure will occur. In our framework, when unexpected events occur at execution, it causes the expected resources to be unavailable, or different ones to be produced. In turn, this causes the agent to realise the step has failed and then re-plan using the new set of available resources.

The Resulting dialogue

The dialogue that is produced by a system can be used to evaluate it on a cognitive basis. It also provides a way to compare different systems. An obvious metric is the length of a dialogue which shows the efficiency of communication between agents.

For the benefit of a human observer we produced natural language using canned text. However the terms communicated between the agents are first order predicates. In comparison with Power's

case study, we avoided the need for a long winded protocol with which agents can communicate plans. Our use of a simpler model of other agents resulted in a much shorter dialogue. Like most systems that focus on only dialogue planning, we did not consider the issue of noisy communication channels or of agents having different ontologies of terms that they communicate. The latter of these issues is discussed in section §10.

9 Conclusion

We have proposed a formal framework for specifying and executing agents that interact through dialogue. This gives a simple logical account for agent reasoning and action, based on the similarity between proofs in ILL and plans. An agent in a given state corresponds to a sequent in the logic. The agent's actions as well as those of other agents, including sensing operations and speech acts, are treated uniformly by specifying them as premises in the sequent. Agents monitor the execution of their plan and re-plan on execution failure.

We presented several case studies in two different domains and used these to evaluate our framework. These studies illustrate our framework's solution to various aspects of dialogue. In particular, we offer a way to design agents that are robust to unexpected events by replanning. Because the order of steps in the interaction protocol is implicit in an agent's knowledge, there is no need for a fixed protocol of turn taking. This allows agents to interact differently in different situations and to deal with interruptions and unexpected questions. In terms of knowledge representation, we gave a treatment for imperfect understanding based on attributes and a general purpose specification for the speech acts of asking and answering questions as well as following instructions. This allowed significant reuse across the domains and thus simplified the problem of specifying the agents.

We implemented our system and case studies in the Linear Logic programming language Lolli which was itself ported to the distributed language Alice. This allows agents to run in a distributed environment and interact over a network where each agent performs planning and execution asynchronously. Agent interaction is synchronised by the protocol of their speech acts.

Using the rich language of ILL as the foundation for planning gives a powerful way to represent agents' knowledge as well as their goals. The issue of efficiency was addressed by limiting the depth of search and engineering agents to contain points of lazy planning. These points then lead to further planning during execution. More generally, this provides an interesting approach to the combinatorial explosion: make the start of a plan and then wait until things are clearer before making the rest of it.

A salient feature of our framework is that it avoids requiring agents to share goals, plans, and representations of the world. Instead agents interact and work together for 'selfish' reasons: an agent makes a plan that involves interaction with other agents based on the agent's internal model of the actions that other agents might take. The complexity of the model of other agents is left up to the agent designer. We suggest that explicit negotiation can be built upon a framework of expectation resources as an implicit reactive protocol. Because our framework gives a simple logical foundation to specifying agents, it also opens the possibility to formally prove properties of agent interaction. Such proofs are an important step towards a dependable vision of the semantic web.

10 Further Work

Our framework assumes that agents have the same ontology for terms which are communicated. To relax this restriction on agent design, the framework could be extended to handle cases of ontology mismatches following McNeill's work on dynamic ontology refinement [31].

In our examples, the internal ontology of an agent is built using extra predicates and matched using unification. For instance, we used (`drink coffee`) to denote that coffee is a kind of drink. This simple but limited ontology mechanism could be replaced by a more expressive framework such as SHOIQ [22]. This kind of extension can be implemented, either at the meta-level, or within the agent’s Linear Logic context using the intuitionistic resources to support the ontology reasoning. This would simplify agent design by making the terms within an agent’s representation smaller and by making the ontology of terms more flexible.

Specifying agents is currently done by an agent designer. An interesting area for further work is to consider how the specification can be automatically synthesised or modified. For example, the generation and consumption of unexpected external resources during execution could be used to update an agent’s action operators. Engineering the knowledge for a large and complex agent would also significantly benefit from such an automated specification.

In terms of implementation, we used a naive planner which causes the planning time to grow very quickly. Implementing a more efficient planner which removes some of the symmetries in the search space would significantly improve performance. One way to do this would be to create a direct implementation of ILL with proof terms rather than an interpreted representation of plans within Lolli. Implementing and experimenting with distributed and parallel execution of plans is another area of further work.

The relatively simple logical account of agents provided by our framework makes reasoning about dialogue systems feasible. For example, the claim that a seller agent never sells an item without getting payment can be proved by analysing the interaction as a set of protocols. Investigating such proofs for our framework, following the work of Osman and Robertson [32] and formal analysis of protocols and security APIs investigated by Steel [43], is thus an exciting avenue of further work.

Adapting ILL to have a suitable form of negation would allow simpler specification of agents than using explicit resources representing negative occurrence. This would avoid the need to resort to implementation tricks when seeking to make the representation as brief as possible.

Other obvious extensions to the work in this paper include connecting the framework to a full dialogue system that includes natural language generation, and considering larger case studies. Our framework was inspired by a cognitive analysis of dialogue and its connections to planning. Investigating our framework as a cognitive model for dialogue is another direction of further work.

Acknowledgements

This work was supported by EPSRC Grant: EP/E005713/1. This article was published in the Journal of Logic, Language and Information, the official published version can be found at:

<http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s10849-008-9079-0>

We would like to thank Mark Steedman, Colin Matheson, Ron Petrick, and the Edinburgh Mathematical Reasoning Group for their helpful comments on this work. We would also like to thank the referees who provided helpful and constructive feedback.

A Further Details of Power’s Door Problem

Asking for Help and Planning to Plan

In order for John to ask for help when he cannot find a plan we added a meta-logical feature: when planning fails at a fixed maximal depth, the extra resource `failed` is added. Then a second attempt to find a plan is made. This can use the `failed` resource which is a precondition to ask for help. The

motivation is that an agent should try to solve a problem before asking for help. The result is that John finds the following plan:

```
- do (ask mary (instruction (goal ...)))
- do (getanswer mary (instruction (goal ...)))
- do (instruction I (goal ...))
- do (answer mary (confirm (step I)))
- do (answer mary (confirm (achieved (goal ...))))
```

where the “...” abbreviates John’s goal, namely (`know (attr john loc in)`).

This plan starts by John asking Mary for help to achieve his goal. Her answer, which he anticipates to be an instruction, will generate an expectation that he does the instruction. This is used up by the action operator `do (instruction I (goal ...))`:

```
do (instruction I (goal G)) !:
  (theyexpect A (confirm (step I)))
  ⊗ (theyexpect A (do (instruction I (goal G))))
  ⇨ (theyexpect A (confirm (step I))) ⊗ G
```

Further planning happens at execution time because following an instruction is expected to directly produce the goal, but in practice more work is needed and thus the plan fails to directly achieve the goal. In summary, our use of ‘optimistic’ action operators is a representational technique to allow planning at execution time.

The final steps in the plan are for John to confirm that he did the instruction and that this achieved his goal. The first confirmation step removes Mary’s expectation for a confirmation of performing the action. The last confirmation uses the expectation of receiving further instructions from Mary. These expectations were produced by asking Mary for help in the first step, and are also a precondition for following her instructions, as well as getting instructions from her.

Getting Asked for Help

When an agent is asked for help, it creates a new resource of the form:

```
theyexpect A (reply (instruction (goal G)))
```

In the case of John’s request for help, Mary gets the new resource with the agent `A` instantiated to `john` and the goal `G` is instantiated to `knows john (attr john loc in)`.

Mary’s initial goal is to be helpful, her plan involves a single action called `wait_and_help`, which when executed waits for an agent to ask her something and generates a resource that indicates she has been helpful. However, when she is asked a question, she gets an expectation resource which she cannot ignore. To remove this expectation she makes the following plan:

```
- do (ask john (qattr john loc))
- do (getanswer john (attr john loc in))
- do (getanswer john (confirm (achieved
  (goal (knows john (attr john loc in)))))
```

It is the last step in this plan that removes the expectation for instruction. The previous steps correspond to asking John where he is and hearing that he already has his goal of being inside. However, John says he is outside which causes Mary to have to replan. We could modify the action of being asked for help so that Mary could directly infer that John’s goal has not been achieved. This would skip Mary’s first plan and result directly in her second plan, shown below.

Giving Instructions

Because John is not in, Mary re-plans, still somewhat optimistically, and asks him if the door is open. If it is, then she can simply instruct him to move. However, he tells her the door is closed. Mary then re-plans once more, to find her final plan:

```
- do (pushto door open)
- do (answer john (instruction (movefrom loc out)))
- follows john (instruction (movefrom loc out))
- do (getanswer john (confirm (step (movefrom loc out))))
- do (getanswer john (confirm (achieved
    (goal (knows john (attr john loc in))))))
```

This involves her opening the door and then telling John to move. At the ontology level, we separate actions that our agent performs from those that are done by another agent. This avoids considering plans where an agent instructs itself. The `follows` predicate is used for steps where another agent follows an instruction, while the `do` predicate indicates actions that involve the instructing agent, in this case Mary.

Mary's final plan succeeds; John is in and Mary has been helpful.

B Implementation

```
% Planner implementation: gives a declarative account
% of planning and execution, including failure. This
% source code is for Lolli. More details can be found at:
% http://dream.inf.ed.ac.uk/projects/dialoguell/

%%% top level %%%
% running an agent...
run :- goal Goal, plan_then_exec Goal.

plan_then_exec Goal :-
    searchdepth D,
    write (planning_up_to_depth D),
    (id_plan P Goal D)
    & (execute Goal P).

%%% execute %%%
% no more plan
execute Goal id :-
    Goal, % check the goal has been achieved
    ignore. % try to ignore irrelevant stuff and stop

% failed to end plan: start planning again
execute Goal id :-
    write "failed to get goal, replanning...",
    plan_then_exec Goal.

% next plan step involves an action
```

```

execute Goal (then ActName Plan):-
  action_spec ActName IPre IUses ICreates EPre EUses ECreates,
  write (executing ActName), write "\n",
  try (EPre, EUses) MaybeFailed MaybeUsed,
  dest_some MaybeUsed Used,
  (Used % undo using up of Used, we only wanted
   % to check if we can do the action
   -o try_act_and_continue MaybeFailed ActName
      Goal Plan IUses ICreates).

% parallel execution is flattened to sequential execution.
execute Goal (par id P2) :-
  execute Goal P2.
execute Goal (par (then A P1) P2) :-
  execute Goal (then A (par P1 P2)).
execute Goal (par (par P1 P2) P3) :-
  execute Goal (par P1 (par P2 P3)).

% execution of conditional plans
execute Goal (case ActName Case1 Plan1 Case2 Plan2):-
  actionop ActName IPre IUses ICreates EPre EUses ECreates,
  write (executing ActName), write "\n",
  try (EPre, EUses) MaybeFailed MaybeUsed,
  dest_some MaybeUsed Used,
  (Used % undo using Used, only check if we can do the action
   -o try_act_and_continue MaybeFailed ActName Goal
      Plan IUses ICreates).

%%% perform an action %%%
% can execute action and do so
try_act_and_continue none ActName Goal Plan IUses ICreates :-
  nl,write (performing ActName),nl,
  do_action ActName ECreates, IUses,
  write (action_created ECreates), nl,
  ((ECreates, ICreates) -o execute Goal Plan).

% cannot perform the action: some needed resource
% or precondition failed, so replan
try_act_and_continue (some Failed) ActName Goal
  Plan IUses ICreates :-
  nl,write "Execution failed, needed: ", write Failed, nl,
  plan_then_exec Goal.

%%% planner %%%
% wrap in iterative deepening search, over
% max length of sequential path through plan;
% this prefers par plans, because they are shorter.
id_plan P Goal Max :- idp P Goal 1 Max.

```

```

id_plan P Goal Max :-
  (write_sans "trying to find given this failure ...", nl,
   failed Goal -o idp P Goal 1 Max).

idp P Goal N Max :- plan_bdd P Goal N.
idp P Goal N Max :- N < Max, M is N+1,
  write_sans "searching at depth ", write M,
  write_sans " of max ", write Max, nl,
  idp P Goal M Max.

plan_bdd id Goal _ :- Goal, ignore.

%% Uncomment this to find plan with parallel steps
plan_bdd (par P1 P2) (G1,G2) N :- % par with depth bound
plan_bdd P1 G1 N,
  plan_bdd P2 G2 N.

plan_bdd (then ActName Plan) Goal N :-
N > 0, M is N-1, % operator decreases depth bound
action_spec ActName IPre IUses ICreates EPre EUses ECreates,
IPre, EPre, IUses, EUses,
((IPre,EPre,ICreates,ECreates) -o (plan_bdd Plan Goal M)).

%%% tools %%%
% we use (some X) and none as an encoding of option types.
dest_some (some A) A.
dest_some none true.

% join two option collections into a single option collection
join_options none none none.
join_options none (some A) (some A).
join_options (some A) none (some A).
join_options (some A) (some B) (some (A, B)).

% try to use up resources: instantiates those used up (Used)
% as well as those which failed to be used (Failed).
% note: backtracking will find multiple identical solutions.
try (A,B) Failed Used :-
  try A FailedA UsedA,
  try B FailedB UsedB,
  join_options FailedA FailedB Failed,
  join_options UsedA UsedB Used.
try true none none.
try A none (some A) :- A.
try A (some A) none.

```

References

- [1] Abramsky, S.: 1993, ‘Computational interpretations of Linear Logic’. *Theoretical Computer Science* **111**(1–2), 3–57.
- [2] Alice: 2007, ‘The Alice manual’. Programming System Lab, Saarland University, 1.4 edition. www.ps.uni-sb.de/alice/manual.
- [3] Austin, J. A.: 1962, *How to Do Things with Words*. London: Oxford University Press.
- [4] Barber, A.: 1997, ‘Linear Type Theories, Semantics and Action Calculi’. Ph.D. thesis, University of Edinburgh.
- [5] Berners-Lee, T., J. Hendler, and O. Lassila: 2001, ‘The Semantic Web’. *Scientific American*.
- [6] Cervesato, I., J. S. Hódas, and F. Pfenning: 2000, ‘Efficient Resource Management for Linear Logic Proof Search’. *Theor. Comput. Sci.* **232**(1-2), 133–163.
- [7] Chu-Carroll, J. and S. Carberry: 1998, ‘Collaborative response generation in planning dialogues’. *Comput. Linguist.* **24**(3), 355–400.
- [8] Chu-Carroll, J. and S. Carberry: 2000, ‘Conflict resolution in collaborative planning dialogs’. *Int. J. Hum.-Comput. Stud.* **53**(6), 969–1015.
- [9] Cohen, P. R. and C. R. Perrault: 1979, ‘Elements of a Plan-Based Theory of Speech Acts’. *Cognitive Science* **3**, 177–212.
- [10] Cresswell, S.: 2001, ‘Deductive Synthesis of Recursive Plans in Linear Logic’. Ph.D. thesis, University of Edinburgh.
- [11] Cresswell, S., A. Smaill, and J. D. C. Richardson: 1999, ‘Deductive Synthesis of Recursive Plans in Linear Logic’. In: *Proceedings of the 5th European Conference on Planning, Durham, UK*, Vol. 1809 of *LNAI*.
- [12] Decker, S., C. A. Goble, J. A. Hendler, T. Ishida, and R. Studer: 2003, ‘A new journal for a new era of the World Wide Web’. *J. Web Sem.* **1**(1), 1–5.
- [13] Dixon, L., A. Bundy, and A. Smaill: 2006, ‘Planning as Deductive Synthesis in Intuitionistic Linear Logic’. Technical Report EDI-INF-RR-0786, School of Informatics, University of Edinburgh.
- [14] Fernández, R. and U. Endriss: 2007, ‘Abstract models for dialogue protocols.’. *Journal of Logic, Language and Information* **16**(2), 121–140.
- [15] Foster, M. E., T. By, M. Rickert, and A. Knoll: 2006, ‘Human-robot dialogue for joint construction tasks’. In: *ICMI '06: Proceedings of the 8th international conference on Multimodal interfaces*. Banff, Alberta, pp. 68–71.
- [16] Girard, J.-Y.: 1987, ‘Linear Logic’. *Theoretical Computer Science* **50**, 1–102.
- [17] Girard, J.-Y.: 1995, ‘Linear Logic: its syntax and semantics’. In: J.-Y. Girard, Y. Lafont, and L. Regnier (eds.): *Advances in Linear Logic*, No. 222 in London Mathematical Society Lecture Notes Series. Cambridge University Press.
- [18] Greenfield, P. M.: 1991, ‘Language, tools, and brain: The ontogeny and phylogeny of hierarchically organized sequential behavior’. *Behavioral and Brain Sciences* **14**(4), 531–551.

- [19] Harland, J. and D. J. Pym: 2003, ‘Resource-Distribution via Boolean Constraints’. *ACM Trans. Comput. Log* **4**(1), 56–90.
- [20] Harland, J. and M. Winikoff: 2004, ‘Agents via Mixed-Mode Computation in Linear Logic’. *Annals of Mathematics and Artificial Intelligence* **42**(1-3), 167–196.
- [21] Hodas, J. S. and D. Miller: 1991, ‘Logic Programming in a Fragment of Intuitionistic Linear Logic’. In: *Proceedings 6th IEEE Annual Symp. on Logic in Computer Science, Amsterdam, The Netherlands, 15–18 July 1991*. New York: IEEE Computer Society Press, pp. 32–42.
- [22] Horrocks, I. and U. Sattler: 2005, ‘A Tableaux Decision Procedure for SHOIQ’. In: *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. pp. 448–453.
- [23] Kopylov, A. P.: 1995, ‘Decidability of Linear Affine Logic’. In: D. Kozen (ed.): *Tenth Annual IEEE Symposium on Logic in Computer Science*. San Diego, California, pp. 496–504.
- [24] Kraft, D., E. Başeski, M. Popović, A. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, T. Asfour, R. Dillmann, S. Kalkan, F. Wörgötter, B. Hommel, R. Detry, and J. Piater: 2008, ‘Exploration and planning in a three-level cognitive architecture’. In: *Proceedings of the International Conference on Cognitive Systems (CogSys 2008)*.
- [25] Küngas, P. and M. Matskin: 2004, ‘Symbolic Negotiation with Linear Logic’. In: J. Dix and J. A. Leite (eds.): *CLIMA IV*, Vol. 3259 of *Lecture Notes in Computer Science*. pp. 71–88.
- [26] Larsson, D. and D. R. Traum: 2000, ‘Information state and dialogue management in the TRINDI dialogue move engine toolkit’. *Natural Language Engineering* **6**, 323–340.
- [27] López, P. and J. Polakow: 2004, ‘Implementing Efficient Resource Management for Linear Logic Programming’. In: F. Baader and A. Voronkov (eds.): *LPAR*, Vol. 3452 of *Lecture Notes in Computer Science*. pp. 528–543.
- [28] Masseron, M.: 1993, ‘Generating plans in linear logic II: A geometry of conjunctive actions’. *Theoretical Computer Science* **113**, 371–375.
- [29] McCarthy, J. and P. Hayes: 1969, ‘Some philosophical problems from the standpoint of artificial intelligence’. In: B. Meltzer and D. Michie (eds.): *Machine Intelligence 4*. Edinburgh University Press.
- [30] McGinnis, J., D. Robertson, and C. Walton: 2005, ‘Protocol synthesis with dialogue structure theory’. In: *AAMAS ’05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA, pp. 1329–1330.
- [31] McNeill, F. and A. Bundy: 2007, ‘Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution’. *IJSWIS* **3**(3), 1–35. Special issue on ontology matching.
- [32] Osman, N. and D. Robertson: 2007, ‘Dynamic Verification of Trust in Distributed Open Systems’. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI’07)*. pp. 1440–1445.
- [33] Petrick, R. P. A. and F. Bacchus: 2002, ‘A Knowledge-Based Approach to Planning with Incomplete Information and Sensing’. In: M. Ghallab, J. Hertzberg, and P. Traverso (eds.): *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*. Menlo Park, CA, pp. 212–221.

- [34] Petrick, R. P. A. and F. Bacchus: 2004, ‘Extending the Knowledge-Based Approach to Planning with Incomplete Information and Sensing’. In: S. Zilberstein, J. Koehler, and S. Koenig (eds.): *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*. Menlo Park, CA, pp. 2–11.
- [35] Power, R.: 1979, ‘The organization of purposeful dialogs’. *Linguistics* **17**, 105–152.
- [36] Rao, A. S. and M. P. Georgeff: 1995, ‘BDI-agents: from theory to practice’. In: *Proceedings of the First Intl. Conference on Multiagent Systems*. San Francisco.
- [37] Robertson, D.: 2004, ‘A Lightweight Coordination Calculus for Agent Systems.’. In: J. A. Leite, A. Omicini, P. Torroni, and P. Yolum (eds.): *DALT*, Vol. 3476 of *Lecture Notes in Computer Science*. pp. 183–197.
- [38] Russell, S. and P. Norvig: 2002, *Artificial Intelligence: A Modern Approach*. London: Prentice-Hall International, 2nd edition.
- [39] Schegloff, E. A. and H. Sacks: 1973, ‘Opening up closings’. *Semiotica* **8**(4), 289–327.
- [40] Steedman, M.: 1997, ‘Temporality’. In: J. F. A. K. van Benthem and G. B. A. ter Meulen (eds.): *Handbook of Logic and Language*. pp. 895–935.
- [41] Steedman, M.: 2002, ‘Plans, Affordances, And Combinatory Grammar’. *Linguistics and Philosophy* **25**(5-6), 723–753.
- [42] Steedman, M. and R. Petrick: 2007, ‘Planning Dialog Actions’. In: *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue (SIGdial 2007)*. pp. 265–272.
- [43] Steel, G.: 2006, ‘Formal Analysis of PIN Block Attacks’. *Theoretical Computer Science* **367**(1-2), 257–270. Special Issue on Automated Reasoning for Security Protocol Analysis.
- [44] Sycara, K., M. Klusch, S. Widoff, and J. Lu: 1999, ‘Dynamic service matchmaking among agents in open information environments’. *SIGMOD Rec.* **28**(1), 47–53.
- [45] Takeda, H.: 2004, ‘Semantic web: a road to the knowledge infrastructure on the internet’. *New Generation Computing* **22**(4), 395–413.
- [46] Traum, D. R. and J. F. Allen: 1994, ‘Discourse Obligations in Dialogue Processing’. In: J. Pustejovsky (ed.): *Proceedings of the Thirty-Second Meeting of the Association for Computational Linguistics*. San Francisco, pp. 1–8.
- [47] Willmott, S., F. O. F. Pena, C. Merida-Campos, I. Constantinescu, J. Dale, and D. Cabanillas: 2005, ‘Adapting Agent Communication Languages for Semantic Web Service Inter-Communication’. In: *WI ’05: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*. Washington, DC, USA, pp. 405–408.