Planning ramifications: When ramifications are the norm, not the 'problem'

Debora Field
Dept. of Computer Science, University of Liverpool
Liverpool L69 3BX, UK
debora@csc.liv.ac.uk

Allan Ramsay
School of Informatics, University of Manchester
PO Box 88, Manchester M60 1QD, UK
allan.ramsay@manchester.ac.uk

Abstract

This paper discusses the design of a planner whose intended application required us to solve the so-called 'ramification problem'. The planner was designed for the purpose of planning communicative actions, whose effects are famously unknowable and unobservable by the doer/speaker, and depend on the beliefs of and inferences made by the observer/hearer. Our fully implemented model can achieve goals that do not match action effects, but that are rather entailed by them, which it does by reasoning about how to act: state-space planning is interwoven with theorem proving in such a way that a theorem prover uses the effects of actions as hypotheses.[1]

Introduction

Seeing the word 'ramification' so often bound to the word 'problem', it is easy to get the impression from the literature that the ramifications of actions are viewed by the AI planning community as an annoying hindrance to their AI planning ambitions. We, however, see ramifications very differently. They are the focus of our planning ambition and the mechanism of its success. Why? Because we are interested in modelling an every-day human activity which is totally dependent upon the ramifications of actions: human-to-human communication.

As far as communication is concerned, each man (and woman) is an island. I have things I want you to believe, and to this end I do my best to make appropriate signs to you—in writing, speech, smoke signals, facial gestures, and so on. You see my signs, and you decide for yourself what they mean. There is nothing I can do to ensure that you receive the message I want you to get. All I can do is make my signs, and put my trust in the ramifications of my actions.

Consider the human, John. Imagine John's current goal is to get human Sally to believe the proposition *John is kind*. John has no direct access to the environment he wishes to affect—he cannot simply implant *John is kind* into Sally's belief state. John knows that Sally has desires and opinions of her own, and that he will have to plan something that he considers might well lead Sally to infer *John is kind*. This means that when John is planning his action—whether to give Sally some chocolate, pay her a compliment, tell her he is kind, lend her his credit card—he has to consider the many different messages Sally might infer from the one thing John chooses to say or do.

To plan communicative acts is, then, to plan actions by taking into account their possible ramifications. How do we do this? We took a backward-chaining theorem prover, and adapted it for hypothetical reasoning about the effects of actions. Our backward-chaining reasoner essentially says, "I could prove this backwards if you allowed me to introduce these hypotheses". The fully implemented model is thus able to plan to achieve goals that do not match action effects, but that are entailed by them. Our planner was developed by first adapting (Manthey & Bry 1988)'s first-order theorem prover, Satchmo, into a theorem prover for a highly intensional logic (Ramsay 2001), namely, a constructive version of property theory (Turner 1987). To this was added a deduction theory of knowledge and belief (Konolige 1986) so that the planner can reason with its beliefs about the world, including its beliefs about others' beliefs. State-space planning (based on foundational work in classical planning (Newell, Shaw, & Simon 1957; Newell & Simon 1963; Green 1969; McCarthy & Hayes 1969; Fikes & Nilsson 1971)) was then interwoven with theorem proving in such a way as to enable planning for entailed goals.

Satchmo

The theorem prover we present was developed by extending Manthey and Bry's (1988) first-order theorem prover, Satchmo (SATisfiability CHecking by MOdel generation). For model generation we convert the standard form to SEQUENT FORM, where a sequent is a formula of the form $\Gamma \Rightarrow \Delta$ where Γ is $\top$, an atomic formula, or a conjunction of atomic formulae, and Δ is $\bot$, an atomic formula, or a disjunction of atomic formulae. Satchmo was designed for carrying out proof by contradiction, where you show that some formula A follows from a set of assumptions α by converting $\alpha \cup \{\neg A\}$ to normal form, and showing that this set

[1]Initially supported by an EPSRC grant, with recent developments partially funded under EU-grant FP6/IST No. 507019 (*PIPS*: Personalised Information Platform for Health and Life Services).

has no models. The goal is to show that the set of sequents obtained from the assumptions α and the negation $\neg A$ of the goal supports a proof of $\perp$, since this means that no model of α can be a model of $\neg A$, and hence that all models of α are models of A (i.e., $\alpha \models A$).

Model generation proceeds by distinguishing between HORN SEQUENTS, and DISJUNCTIVE SEQUENTS. A Horn sequent has a single literal as its consequent; a disjunctive sequent has a disjunction as its consequent. The proof proceeds in two stages:

(MG-i) Try to prove $\neg A$ by backward chaining among the Horn sequents.

(MG-ii) If this fails, find a disjunctive sequent whose antecedent can be proved using the Horn sequents, but whose consequent cannot. Add each disjunct from the consequent in turn, and try again. The point here is that if we know Γ and $\Gamma \Rightarrow A_1 \vee A_2$, then we know that either A_1 or A_2 must hold. So if we can show that $\perp$ follows from what we know already plus either of A_1 or A_2, then we know that $\perp$ actually follows from what we know already.

A constructive epistemic intensional logic

The original presentation of Satchmo is unsuitable for our purposes, since it assumes a classical version of predicate logic, whereas we use a constructive epistemic logic.

An epistemic logic is necessary because the program needs to be able to do reasoning about what agents believe, including what they believe about the beliefs of other agents. The theorem prover embodies a deduction model of belief (Konolige 1986), rather than a 'possible worlds' model (Hintikka 1962; Kripke 1963). In a deduction model, agents are allowed to have sets of beliefs that are incomplete, inaccurate, and inconsistent, to have imperfect inference strategies, and to *do* inference—which is much more useful for modelling human epistemic reasoning than an agent who is logically omniscient and automatically knows everything that follows from his belief set.

We use a constructive logic, because we consider it is essential for our purpose—to model natural language—as we will now briefly argue. Our argument falls under two themes: (i) We need a theorem prover that can reason with formulae in which propositions are quantified over; (ii) We need a theorem prover that more closely models the way *people* do reasoning, including reasoning about a proposition that a person believes to be false.

A theorem prover for property theory

There are a number of phenomena in natural language that we can cope with if we are allowed to quantify over propositions, but that seem otherwise very hard to capture. Consider the word '*only*'. If S says to H:

I only <u>touched</u> it

(with a voiced stress on *touched*), S is inviting H to compare the action A that S *did* do (*touched*) with some other action A' that S did *not* do, and which is in some way stronger than A (*broke*, perhaps). To represent the meaning of '*only*' we require intensional meaning postulates—intensional, because they comment on the truth or falsity of other parts of the sentence. For '*only*' we need something like:

$$\forall P \forall X(only(P, X) \Rightarrow (P.X \& \exists P'(similar(P, P') \& \neg P'.X)))$$

This axiom says that if $only(P, X)$ holds, where P is some property and X is an arbitrary entity (possibly a property itself), then P does hold of X, but there is some property P' which is similar to P, and which you might have expected to hold of X, but which in fact fails to do so. So then, to adequately represent the meaning of even a common little word like '*only*', not to mention many other natural language constructions, we need to be able to quantify over propositions (Ramsay 1994).

Quantifying over propositions, however, opens the way to the paradoxes of negative self-reference—Russell's set, the Epimenedes paradox, Grelling's paradox, and so on. Therefore, we need some way of preventing this that still admits common natural language usage (e.g., the many self-referring pronouns: '*this*', '*me*', '*himself*' ...), which the earlier classical solutions to this problem do not (e.g., Whitehead and Russell's (1910) THEORY OF TYPES, and MONTAGUE SEMANTICS (Thomason 1974)).

Turner's PROPERTY THEORY (1987) provides a solution to this problem by allowing you to say whatever you want, but then placing constraints on what can be proved. Turner's analysis involves taking a classical treatment of first-order logic, and adding λ-abstraction and β-reduction to it (or at any rate, operations which look extremely like λ-abstraction and β-reduction). It allows propositions and properties to occur as arguments, and thus provides the expressive power we need. It provides an operator for constructing terms corresponding to propositions and properties, so that they can appear in the positions where terms are required (i.e., as predicate arguments), together with a series of predicates for retrieving the truth conditions of such a term. The main advantage of Property Theory over languages like Montague Semantics is that it is a type-free language, and hence provides considerable extra expressive power. Spelling out the meaning of a word like '*only*' requires you to produce *untyped* intensional meaning postulates—untyped because this word can apply to phrases of (almost) any syntactic type, so that their semantic analysis must apply to phrases of almost any semantic type.

Turner's Property Theory is, then, attractive for our natural language purposes. Unfortunately, (Herzberger 1982) shows that it is impossible to provide a normal form for

Turner's version of the logic, and hence it is very difficult to implement a theorem prover for it. Our solution to this is to take a constructive treatment of first-order logic, allow unrestricted use of both λ-abstraction and β-reduction, and avoid the paradoxes by placing constraints on the assumptions that can be used in a well-founded proof (Ramsay 2001).

Modelling human reasoning

The way humans do every-day reasoning is, we consider, quite different from the way reasoning is handled under classical logic. In classical logic, for example, and using our general knowledge, we judge the following formulae to be true:

(1) Earth has one moon ⇒ Elvis is dead
(2) Earth has two moons ⇒ Elvis is alive
(3) Earth has two moons ⇒ Elvis is dead

(1) is true simply because antecedent and consequent are both true formulae. We find this truth odd, however, because of the absence of any discernible relationship between antecedent and consequent. (2) and (3) are true simply because the antecedent is false, which seems very counter-intuitive. Even more peculiarly, the following formula is provable in classical logic in all circumstances:

(4) (Earth has one moon ⇒ Elvis is dead) or
(Elvis is dead ⇒ Earth has one moon)

but it feels very uncomfortable to say that it must be the case that one of these implies the other.

In order to avoid having to admit proofs like this, and to be able to do reasoning in a slightly more human-like way, we choose constructive logic and natural deduction. In order to prove *P* ⇒ *Q* by natural deduction, one must show that *Q* is true *when P* is true; if *P* is not true, constructive logic does not infer *P* ⇒ *Q*. This treatment of implication hints at a relationship between *P* and *Q* which is absent from material implication.

Taking a constructive view also allows us to simplify our reasoning about when the hearer believes something of the form $P \Rightarrow Q$, and hence (because of the constructive interpretation of $\neg P$ as $P \Rightarrow \bot$) about whether she believes $\neg P$. We will assume that $believes(H, P)$ means that H *could* infer P on the basis of her belief set, not that she already does believe P, and we will examine the relationship between $believes(H, P \Rightarrow Q)$ and $believes(H, P) \Rightarrow believes(H, Q)$.

Consider first $believes(H, P) \Rightarrow believes(H, Q)$. Under what circumstances could you convince yourself that this held?

For a constructive proof, you would have to assume that $believes(H, P)$ held, and try to prove $believes(H, Q)$. So you would say to yourself 'Suppose I were H, and I believed P. Would I believe Q?' The obvious way to answer this would be to try to prove Q, using your own rules of inference. Suppose you came up with such a proof. Assuming that H's rules of inference were the same as yours, you would then be able to assume that she could also carry out this proof. But if that were the case, then she would have a proof of $P \Rightarrow Q$ available to her, and hence it would be reasonable to conclude $believes(H, P \Rightarrow Q)$.

Suppose, on the other hand, that you believed $believes(H, P \Rightarrow Q)$, and that you also believed $believes(H, P)$. This would mean that you thought that H had both $P \Rightarrow Q$ and P available to her. But if you had these two available to you, you would be able to infer Q, so since H is very similar to you, she should also be able to infer Q. So from $believes(H, P \Rightarrow Q)$ and $believes(H, P)$ we can infer $believes(H, Q)$, or in other words, $(believes(H, P \Rightarrow Q)) \Rightarrow (believes(H, P) \Rightarrow believes(H, Q))$.

We thus see that if we take $believes(H, P)$ to mean 'If I were H I would be able to prove P', then $(believes(H, P \Rightarrow Q))$ and $(believes(H, P) \Rightarrow believes(H, Q))$ are equivalent. This has considerable advantages in terms of theorem proving, since it means that much of the time we can do our reasoning by switching to the believer's point of view and doing perfectly ordinary first-order reasoning. If, in addition, we treat $\neg P$ as a shorthand for $P \Rightarrow \bot$, we see that $believes(H, \neg P)$ is equivalent to $believes(H, P) \Rightarrow believes(H, \bot)$. If we take the further step of assuming that nobody believes $\bot$, we can see that $believes(H, \neg P) \Rightarrow \neg believes(H, P)$ (though not $\neg believes(H, P) \Rightarrow believes(H, \neg P)$). We cannot, however, always assume that everyone's beliefs are consistent, so we may not always want to take this further step (note that in possible worlds treatments, we are *forced* to assume that everyone's beliefs are consistent), but it is useful to be able to use it as a default rule, particularly once we understand the assumptions that lie behind it.

Constructive Satchmo

We have said that the original presentation of Satchmo is unsuitable for our purposes, since it assumes a classical version of predicate logic. This means that you can prove P by showing that $\neg P$ is unsatisfiable, and you can also use equivalences such as $((P \Rightarrow Q) \Rightarrow R) \iff ((Q \Rightarrow R) \& (P \ or \ R))$, which are not available in constructive logic. We therefore need to adapt Satchmo so that it works properly for constructive logic, and so that it can handle epistemic information.

We do this in two stages: first we have to convert our problem into an appropriate normal form, and then we have to adapt the basic Satchmo engine to work constructively with this normal form. Epistemic information is represented by using Wallen's (1987) notion of CONTEXT, which in our case is epistemic context.

The construction of a normal form proceeds in three stages.

(i) We start by making very straightforward textual changes, to make standard logical form look a bit more like Prolog, and to get rid of existential quantifiers.

- NF-1 Replace `(A & B)` by `(A', B')` and `(A or B)` by `(A'; B')`, where `A'` and `B'` are the normal forms of `A` and `B`.
- NF-2 Replace `not(A)` by `(A' ⇒ absurd)`.
- NF-3 Replace `P ⇒ (Q ⇒ R)` by `((P & Q) ⇒ R)'`.
- NF-4 Replace `believes(J,A)` by `A@@[believes(J)]`. (See below for further discussion of epistemic contexts)
- NF-5 Skolemise away existential quantifiers, and remove all universal quantifiers.

(ii) Separate the result of (i) into Horn and non-Horn clauses, and convert the Horn clauses to ordinary Prolog.

- PL-1 If the normal form of `P` is atomic then assert it as a Prolog fact.
- PL-2 If the normal form of `P` is `(Q , R)` then deal with `Q` and `R` individually.
- PL-3 If the normal form of `P` is `(Q ; R)` then assert `split(Q ; R)` as a Prolog fact.
- PL-4 If the normal form of `P` is `(K ⇒ Q)` where `Q` is atomic then assert `Q :- K` as a Prolog rule.
- PL-5 If the normal form of `P` is `(K ⇒ (Q , R))` then deal with `(K ⇒ Q)` and `(K ⇒ R)` individually.
- PL-6 If the normal form of `P` is `(K ⇒ (Q ; R))` then assert `split(Q ; R) :- K` as a Prolog rule.
- PL-7 If the normal form of `P` is `(A ⇒ B) ⇒ C` then assert `C :- (A => B)` as a Prolog rule. See (MG-3) for further discussion of this rule.

(iii) Perform any optimisations that you can on these.

- OP-1 Remove all 'pure literals' (Kowalski 1975) from the clause set, and store for easy restoration if they later become 'impure'.
- OP-2 Satchmo can be made to perform very poorly if you include disjunctive clauses where one or both disjuncts is actually irrelevant, so optimise relevance by banning the use of a `split` clause until it has been shown that its consequents *will* contribute to a proof of `G`. Do this by asserting a rule which will itself add the `split` clause when its consequents have been shown to be relevant.
- OP-3 Include a 'loop checking' procedure which ensures that any loops which might arise in the automatically generated model are checked (see later for how this is done).
- OP-4 Cut out unnecessary multiple proofs of the same goal. If the head of a rule is ground at the point when called, we know there is no point in finding multiple proofs of it, so place a Prolog cut, to be invoked iff the head was ground at the point when the clause was called.

Once we have the problem converted to normal form, we can use the following adaptation of the basic model generation algorithm.

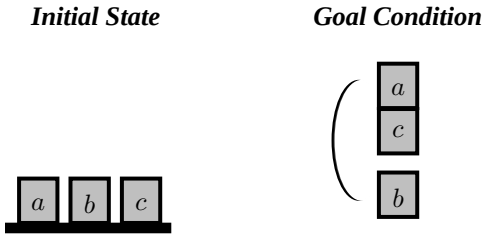
- MG-1 If you can prove `A` by using Prolog facts and rules then you can prove it.
- MG-2 You can prove `A` if you can prove `split(P , Q)` and any of (i) `P ⇒ A` and `Q ⇒ A`, or (ii) `P ⇒ A` and `not(Q)`, or (iii) `not(P)` and `Q ⇒ A`.
- MG-3 To prove `A ⇒ B`, assert `A` and try to prove `B`. If asserting `A` allows you to prove `B` then you have a proof of `A ⇒ B`. Whether or not you succeed in proving `B`, you must retract `A` afterwards.

(MG-1) and (MG-2) are exactly as in the original presentation of Satchmo, except that since Satchmo works by trying to show that the hypotheses + the negation of the goal are unsatisfiable, it always tries to prove `absurd`, whereas a constructive version has to show that the goal itself is provable from the hypotheses (though the second pair of routes through (MG-2) allow you to do this slightly indirectly). (MG-3)[2] is introduced because Satchmo relies on the classical equivalence between $((P \Rightarrow Q) \Rightarrow R)$ and $((Q \Rightarrow R)\&(P\ or\ R))$ when constructing normal forms. This equivalence is no longer available: if we want to prove $P \Rightarrow Q$ we have to use $\Rightarrow$-introduction. In particular, if you want to prove `not(P)`, as you might as a consequence of using (MG-2(ii)) or (MG-2(iii)), then you will have to do this by using (MG-3) to prove `P ⇒ absurd`. There follows a skeletal implementation of this.

```
% You can prove A either directly
prove(A):-
    A.
% or by proving (P or Q),(P => A)and(Q => A)
prove(A):-
    split(P;Q),
    % Check you haven't tried this already
    \+ (P;Q),
    prove(P => (A or absurd)),
    prove(Q => (A or absurd)).
% To prove (P => A), assert P
% and try to prove A (with some funny
% bookkeeping to tidy up after yourself)
(P => A):-
    assert(P),
    (prove(A) -> retract(P);
    (retract(P), fail)).
```

[2] MG-3 is equivalent to $\Rightarrow$-introduction from standard constructive logic. If you start by checking that `B` is not provable without `A`, you get RELEVANT IMPLICATION (Anderson & Belnap jr. 1975).

Figure 1:



The key non-cosmetic differences between this and Satchmo are that (i) this version implements a constructive version of first-order logic rather than a classical one, and (ii) it is slightly more direct when faced with clauses of the form $((P \Rightarrow Q) \Rightarrow R)$. Most of the work in Satchmo is performed in the backward chaining phase where the Prolog facts and rules are being used to prove specific goals. By converting $((P \Rightarrow Q) \Rightarrow R)$ to `R :- (P ⇒ Q)`, we ensure that this rule is activated when it is required, at the cost of having to prove `P ⇒ Q` by asserting `P` and showing that `Q` follows from it. If we convert $((P \Rightarrow Q) \Rightarrow R)$ to `R :- (P ⇒ Q)` and `split(R ; P)`, we end up having to explore the consequences of asserting `P` anyway.

Planning by hypothesising actions

Now that the theorem prover has been introduced, let us move on to how the theorem prover is used to hypothesise actions in order to achieve goals that do not match action effects, but that are entailed by them. First, a 'blocks world' example will be given to illustrate what we mean by 'planning ramifications'. Following this will be an explanation of how information concerning which actions have been hypothesised is carried around in the model. Finally, a description of the model's planning procedure will be given.

An example

Figure 2 shows a list of facts describing a simple knowledge state, which asserts: that j knows (strongly believes) there is a certain configuration of blocks (three, each on the table); that everyone knows about two actions: '*stack*' and '*unstack*'; and that everyone knows that '*above*' is the transitive closure of '*on*'.

The *Initial State* in Figure 1 is a pictorial representation of the knowledge state shown in Figure 2, while the *Goal Condition* in Figure 1 pictures the following proposition:

```
believes(j, above(a,b) & on(a,c))
```

This is the plan solution returned when we call the planner to achieve this proposition from the initial state described by `state_1`:[3]

[3]For clarity's sake, a number of details are missing from Figure 2 which are present in the model, including constraints on the recursive `above(_,_)` rule and on the effects of actions.

Figure 2:

```
state_1(
 believes(j, isblock(a) & isblock(b)
  & isblock(c) & on(a,table) & on(b,table)
  & on(c,table) & clear(a) & clear(b)
  & clear(c))
 & forall(Y,
    believes(Y,
     action(doer(slave),stack(slave,F,G),
      precons(isblock(G) & isblock(F)
       & on(F,table)
       & clear(G) & clear(F)),
      add(on(F,G)),
      delete(on(D,table) & clear(G)))))
 & forall(Y,
    believe(Y,
     action(doer(slave), unstack(slave,F,G),
      precons(on(F,G) & clear(F)
       & isblock(F) & isblock(G)),
      add(clear(G) & on(F,table)),
      delete(on(F,G)))))
 & forall(Y,
    believes(Y,
     (on(P,Q) => above(P,Q))))
 & forall(Y,
    believes(Y,
     ((on(R,Q) & above(P,R))
      => above(P,Q)))).
```

```
[stack(slave,c,b),stack(slave,a,c)]
```

Disregarding epistemic matters for the moment, in order to achieve *above(a,b) & on(a,c)*, something different from an action with an *above(_,_)* expression in its add list is needed (note that although the goal contains a predicate of the form *above(_,_)*, the add list of action *stack* does not contain any *above* predicates). Placing *a* onto *b*, for example, will make *above(a,b)* proveable, but it will also make the achievement of *on(a,c)* impossible. By reasoning with the rules that describe the meaning of *above* as the transitive closure of *on*, the theorem prover hypothesises that the proposition *on(a,X) & on(X,b)* might enable the proof of *above(a,b)* to be completed. It also knows that *on(X,Y)* is an effect of action *stack(X,Y)*. A proof of the preconditions of action *stack(a,X)* is invoked by the planning search, and the process continues (with backtracking), until a full plan solution is found. Thus the planner is able to find *stack* actions to achieve *above* goals, because it knows that the effects of *stack(X,Y)* include *on(X,Y)*, while also knowing that the ramifications of *stack(X,Y)* include *above(X,***Z***)*.

Labels

Now we come to explain how information concerning which actions have been hypothesised is carried around in the model. As a starting point for this, we refer again to the theorem prover's loop-checking mechanism. The programs we are concerned with are generated automatically from sets

of statements, and it is not appropriate to restrict what can be said in this language, just because we are worried about the theorem prover getting into a loop. In order to prevent infinite loops of reasoning occurring, we include an abbreviated copy of the proof stack in a 'label' (after (Gabbay 1996)). The label carries non-logical, arbitrary information about the progress of a proof, and is used for a variety of purposes. Labels are threaded through the clause, so that information can be passed from one subgoal to the next. For loop checking we add the current goal to the goal stack at the points where we are about to call something which might lead us into a loop, and we start this call by explicitly looking to see whether we are in a loop.[4] Another use for the labels is to carry around the equivalence classes that result from using rules relating to equality, which we use for dynamically rewriting clauses (rewriting the entire clause set, as proposed by (Gallier *et al.* 1993), is very expensive if you have large clause sets: we prefer to rewrite clauses as we use them, using information encoded in the label).

The two most important entries in the label (as far as this paper is concerned) are (i) the hypothetical actions that the theorem prover collects, and (ii) a note of the epistemic context of the proposition to which the label belongs. Referring again to the blocks-world example, Figure 3 shows `above(a,b)`, with its label, at the point in the search where the theorem prover has identified some actions that would achieve it.

The label is carried around as the first argument of the goal predicate `above`. The arguments `a` and `b` of the goal are the last arguments of the label (lines (32) and (33)). The normal form `above(a,b)@@[believe(j)]` of the goal is carried around in the label for goal protection purposes, and can be seen as an argument of `top` in line (2). The stack for the loop checker is `trail(C)`, seen on line (3). The actions `stack(slave,E,b)` and `stack(slave,a,E)` that have been found can be seen in lines (7 ff) and (20 ff) as arguments of `hypotheses(...)`. The epistemic context `[believe(j)]` of the goal is seen in line (31). Let us say a brief word here about epistemic entailment.

Epistemic entailment During a proof, a call is made to procedure `checkContexts` to see whether the epistemic context of a proposition is entailed by the epistemic context of the proposition from which it is being currently proved. The KNOWLEDGE AXIOM, TRANSMISSIBILITY AXIOM, and an INTROSPECTION AXIOM (a two-way reading of the so-called POSITIVE INTROSPECTION AXIOM (Hintikka 1962)) hold for knowledge as

[4]Clearly, the definition of `checkloop` is critical here—if you define it too tightly you will lose proofs which are actually available; if you define it too loosely, you will still get into loops. For practical purposes, we choose to define it quite tightly, so that we have some confidence that the algorithm will terminate (it is therefore, of course, bound to be incomplete. That's a choice you have to make—the more loops you cut out, the more potentially provable theorems you miss).

Figure 3:

```
 1. above(label(shared(refs(B),
 2.   top(above(a, b) @@ [believe(j)])),
 3.     nonshared(trail(C),
 4.       D,
 5.       hypotheses([{,(above(a, b)
 6.         @@ [believe(j)],
 7.         (action(doer(slave),
 8.           stack(slave, E, b),
 9.           precons(isblock(b)
10.                   & isblock(E)
11.                   & on(E,table)
12.                   & clear(b)
13.                   & clear(E)),
14.           add(on(E, b)),
15.           delete(on(E, table)
16.                   & clear(b))),
17.         [believe(j)]))},
18.       {,(above(a, b)
19.         @@ [believe(j)],
20.         (action(doer(slave),
21.           stack(slave, a, E),
22.           precons(isblock(E)
23.                   & isblock(a)
24.                   & on(a,table)
25.                   & clear(E)
26.                   & clear(a)),
27.           add(on(a, E)),
28.           delete(on(a, table)
29.                   & clear(E))),
30.         [believe(j)]))}]),
31.       context([believe(j)]))),
32.  a,
33.  b)
```

modelled by the program. The axioms that hold for belief are the INTROSPECTION AXIOM and a mutual belief axiom.

Mutual beliefs are beliefs that conversants believe that they share. For example, we represent '*John believes that he and Sally mutually believe that pigs can't fly*', in standard logic as:

```
believes(john,
  believes(sally,
    mutuallybelieve([john,sally]),
     not(fly(pigs))))
```

A mutual belief that p by j and s entails an infinite number of beliefs ($B_j p$, $B_s p$, $B_j B_s p$, $B_s B_j p$, $B_j B_s B_j p \ldots$), however, this does not present a difficulty for the theorem prover, because, as mentioned above, the theorem prover embodies a deduction model of belief.

Now we will discuss in more detail how the actions got into the label, and how they are used to derive a solution to a planning problem.

Procedure

It is difficult to decide which is the more helpful way to describe our model. It is both a planner that employs a theorem

prover to hypothesise actions, and it is a theorem prover that employs a planning search. It is perhaps more intuitive to depict the model as the former, because the user calls the planning search, and the planning search calls the theorem prover; however, since the theorem prover hypothesises actions (a single action, or a series of actions), there is clearly a fuzzy boundary between theorem proving and planning in the model.

What we can say is that whereas the theorem prover posits desirable series of actions, it does not of itself invoke a search to find out whether the actions are doable (and to make them doable if they are not); this testing of and planning for preconditions is carried out by the planning search (which again invokes the theorem prover to hypothesise desirable actions). Here is a more formal and more detailed explanation.

The user calls `plan` to return a plan that would achieve some goal G from some initial state $W0$ (which has already been asserted into the Prolog database), and `plan` calls the theorem prover.

The theorem prover first reasons to see whether $W0 \models G$. It ensures no plan search is carried out by insisting that the argument of `hypotheses` (carried in the label) be the empty list. If $W0 \models G$, the empty plan is returned to the user as the solution. Otherwise, the theorem prover is called a second time, this time with a variable as the argument of `hypotheses`.

Now the theorem prover distributes the epistemic context of G over the conjuncts of G. We will illustrate using our blocks-world example, so goal G

```
(above(a,b) & on(a,c))@@[believes(j)]
```

becomes G'

```
above(a,b)@@[believes(j)]
& on(a,c)@@[believes(j)]
```

Next, the first conjunct `above(a,b)@@[believes(j)]` of G' is addressed, and an action is identified whose effects entail it. How is this done? Addressing conjunct `above(a,b)@@[believes(j)]` of G', the theorem prover finds that `above(a,b)` is the consequent of a rule whose antecedent is `on(a,b)`, and whose epistemic context is `[believes(Y)]` (see Figure 2). The epistemic context is distributed over the rule,[5] and a proof is sought of antecedent `on(a,b)@@[believes(Y)]`.

Now the theorem prover is looking for an action whose effects would render `on(a,b)@@[believes(Y)]` true. Individual action effects are stored as 'hypothetical facts' in the Prolog database. Figure 4 shows a skeletal version of the effect `on(F,G)` of action `stack(slave,F,G)` as it is stored in the database:[6]

[5] See earlier ('Modelling human reasoning') for our justification of this move.

[6] In order to make our presentation as clear as possible for the reader, we have missed out much of the body of the rule presented in Figure 4. Present in the actual body are procedures for checking whether the head of a rule is ground, for loop checking, and for managing equivalence classes (all mentioned earlier), as well as some additional constraints on `stack`. The label is also missing a lot of arguments, most of which have been mentioned already. The preconditions list has also been simplified.

Figure 4:

```
0. on(label(..., hypotheses(D),
1.          context(E)),
2.    F, G) :-
3.   checkContexts([believes(H)],E),
4.   hypothesis(
5.         label(hypotheses(D),
6.           context([believes(H)])),
7.         action(doer(slave),
8.             stack(slave,F,G),
9.             precons(isblock(G)
10.                      & isblock(F)
11.                      & on(F,table)
12.                      & clear(G)
13.                      & clear(F)),
14.            add(on(F,G)),
15.            delete(on(F,K)&clear(G))).
```

This hypothetical fact says "`on(F,G)` *would* be true, if you allowed me to introduce these hypotheses". The hypothesis the theorem prover wants to introduce here is `stack(J,F,G)` (lines (4)–(15)).

Continuing with the example, the hypothetical fact `on(F,G)` is unified with our current subgoal `on(a,b)`, and (among other things) a check is made to see whether the epistemic context `[believes(j)]` of the subgoal is entailed by the epistemic context `[believes(H)]` of the action, which it is (see lines (1), (3), and (6)). Now the label of `on(a,b)` is threaded through into the label of `above(a,b)`, carrying with it the list of hypothetical facts (which contains one fact at the moment in our example), and the list is returned to `plan`. Now the epistemic context of the action `stack(slave,a,b)` is distributed over the action's preconditions, and `plan` is called to prove the (now epistemic) preconditions of `stack(slave,a,b)`. Thus the whole planning procedure starts again from the beginning, this time with the preconditions of a desirable action as the goal. In our example, all the preconditions of `stack(slave,a,b)` are true, and so the action `stack(slave,a,b)` is applied (the `add` list facts are added to the Prolog database, and the `delete` list facts are deleted). A goal protection check is made, which succeeds, and the next conjunct `on(a,c)@@[believes(j)]` of the goal G' is addressed.

The procedure continues for the second goal conjunct much as just described for the first, except this time there is no need to invoke an `above(_,_)` rule to find a desirable action. However, the goal protection check fails, because to achieve `on(a,c)@@[believes(j)]`,

the planner would have to 'unachieve' the earlier achieved goal `above(a,b)@@[believes(j)]`. Consequently, Prolog fails and backtracks to try an alternative proof of `above(a,b)`. This time the theorem prover collects two hypothetical facts in its list of hypotheses, and returns them to `plan`. Figure 3 above shows the model's representation of the goal `above(a,b)@@[believes(j)]` at this stage, when the goal has just been returned to `plan` along with the hypothetical facts that would enable its proof to be completed. Notice that `stack(slave,E,b)` and `stack(slave,a,E)` are included as arguments of `hypotheses(...)`, being the two actions that need performing before the proof of `above(a,b)@@[believes(j)]` can be completed.

Having more than one action to deal with this time, `plan` makes a plan to achieve the preconditions of the first action `stack(slave,E,b)`, applies that plan, does a goal protection check, and then repeats the procedure for the second action `stack(slave,a,E)` (which by now is fully instantiated), with backtracking, until a plan is found that will achieve the first goal conjunct `above(a,b)@@[believes(j)]` from $W0$. The growing plan by this stage is `[stack(c,b),stack(a,c)]`. Now the second conjunct `on(a,c)@@[believes(j)]` of the goal G' is addressed, and the theorem prover finds that $W1$ models `on(a,c)@@[believes(j)]`, so a plan solution has been found.

Comments on procedure

The example that has been given is a simple task, chosen to keep the necessary explanation to a minimum. One factor that makes it simple is that there are only two conjuncts in the goal condition, and only one of these is an `above(_,_)` goal. The planner can, however, achieve goals having many conjuncts, and including many `above(_,_)` goals, including those which require multiple recursive calls of the `above(_,_)` rules in order to be achieved.

Another factor that makes the example simple is that, due to the fact that all the blocks are on the table and clear in the initial state, no planning is required to achieve the preconditions of the hypothesised actions. The example is not typical, however, and there are many tasks the planner can achieve which require preconditions to be achieved, not simply proved true.

The simplicity of the example has made it unnecessary to describe aspects of the plan search that have been specially designed to overcome unhelpful goal interactions (as first discussed by (Sussman 1974; Sacerdoti 1975)). These include a cross-plan-splicing procedure we call 'Think Ahead' (Field & Ramsay 2004). It works by incorporating thinking about the preconditions of chronologically later actions into the planning of earlier actions, which it does by exploiting the chronological information in the antecedent of a recursive domain-specific inference rule.

The reader may have noticed that our actions have an explicit 'doer' who is always 'slave':

```
forall(Y,
 believes(Y,
  action(doer(slave),stack(slave,F,G)...)))
```

This is done because we want to maintain a clear distinction between three agent types: (i) the agent who *knows about* actions and does all the thinking (`Y` in the `stack` operator); (ii) the agent who will supposedly *execute* the action (`slave` in `stack`); and (iii) the agent who *observes* the action. No explicit observer is mentioned in the `stack` operator, because we have represented stacking as a non-communicative action, which is generally the case.[7] Our operators for communicative actions, however, require both speaker (doer) and hearer (observer) to be made explicit, so that preconditions lists can refer to both the speaker's beliefs and the hearer's beliefs.

General comments and further work

The reader may have observed that there are perhaps alarmingly large amounts of quoted Prolog in this paper. Although this may appear unorthodox, we feel it serves our purpose well to include the code, and so we hope that the reader will forgive us. We are aiming to exploit the efficiency of Prolog as an engine for backward chaining through Horn clauses. The inclusion of a substantial amount of Prolog in the discussion is intended to show how we can obtain directly executable Prolog from problems stated in epistemic logic.

The authors do recognise that most of the AI community views ramifications as a problem (a part of the FRAME PROBLEM (McCarthy & Hayes 1969)), and not the norm. We concede that, whereas our presented model makes plans to achieve what we might call 'positive ramifications' (new propositions which become provable in the new state, but which are not listed in an operator's add list), there has been no mention thus far of what we might call 'negative ramifications'—facts in the knowledge base which are rendered false by the application of an action, but which do not appear in the action operator's delete list, and which therefore remain in the knowledge base, leading to potential inconsistencies.

Currently no procedure is implemented in the model to deal with negative ramifications, because there is no need for it. This is because the inferences made by the entertainment of hypotheses are only held at run-time, they are not collected into a cache to be asserted in the knowledge base for later reference. We recognise that there may be an argument for cacheing inferences, annotated by the hypotheses that were being entertained at that point. However, in our

[7] It is, however, easy to think of situations in which stacking a block is done to convey a message, for example, a mother stacking a block to communicate to her baby how stacking is done. In fact, we consider most 'physical' actions as potential communicative actions.

natural language domain, the costs are likely to outweigh the benefits, because we very seldom repeat inferences under the same sets of assumptions. If we *were* to cache inferences, we would probably appeal to the EXTENDED STRIPS ASSUMPTION (Reiter 1978, p. 407) (formulae not in the delete list of an operator will remain true after the action's performance, unless it can be shown otherwise), and employ a reasoning maintenance system (after (Doyle 1987)). At the point of the addition of a cache of inferences to the knowledge base, the RMS would use necessary supporting conditions to identify and delete any propositions in the database which the new propositions contradicted.

References

Allen, J.; Hendler, J.; and Tate, A. 1990. Editors. *Readings in planning*. San Mateo, California: Morgan Kaufmann.

Anderson, A. R., and Belnap jr., N. 1975. *Entailment: the Logic of Relevance and Necessity*, vol. 1. New Jersey: Princeton University Press.

Doyle, J. 1987. A truth maintenance system. In (Ginsberg 1987), pp. 259–79.

Feigenbaum, E. A., and Feldman, J. 1995. Editors. *Computers and thought*. Cambridge, Massachusetts: MIT Press. First published in 1963 by McGraw-Hill Book Company.

Field, D., and Ramsay, A. 2004. How to build towers of arbitrary heights, and other hard problems. In *Proc. 23rd Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG)*, 20–21 December 2004, Cork, Ireland.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2: 189–208.

Gabbay, D. M. 1996. *Labelled deductive systems*. Oxford University Press: Oxford.

Gallier, P.; Narendran, P.; Plaisted, D.; Raatz, S.; and Snyder, W. 1993. An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the Association for Computing Machinery* 40(1): 1–16.

Ginsberg, M. L. 1987. *Readings in nonmonotonic reasoning*. Los Angeles, California: Morgan Kauffmann.

Green, C. 1969. Application of theorem proving to problem solving. In *Proc. 1st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 219–39.

Herzberger, H. 1982. Notes on naive semantics. *Journal of Philosophical Logic* 11: 61–102.

Hintikka, J. 1962. *Knowledge and belief: An introduction to the two notions*. New York: Cornell University Press.

Konolige, K. 1986. *A deduction model of belief*. London: Pitman.

Kowalski, R. 1975. A proof procedure using connection graphs. In *Journal of the Association for Computing Machinery* 22(4): 572–595.

Kripke, S. 1963. Semantical considerations on modal logic. In *Acta Philosophica Fennica* 16: 83–94.

Manthey, R., and Bry, F. 1988. Satchmo: a theorem prover in Prolog. *Proc. 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Artifcial Intelligence*, pp. 415–434, Berlin: Springer-Verlag.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* 4: 463–502.

Newell, A., and Simon, H. A. 1963. GPS, a program that simulates human thought. In (Feigenbaum & Feldman 1995), pp. 279–93.

Newell, A.; Shaw, J. C.; and Simon, H. A. 1957. Empirical explorations with the logic theory machine. In *Proc. Western Joint Computer Conference* 15: 218–239.

Ramsay, A. M. 1994. Focus on "Only" and "Not". In Y. Wilks. Editor. *Proc. 15th International Conference on Computational Linguistics (COLING-94)*, Kyoto, pp. 881–885.

Ramsay, A. 2001. Theorem proving for untyped constructive λ-calculus: implementation and application. In the *Logic Journal of the Interest Group in Pure and Applied Logics*, Vol. 9(1): 83–100.

Reiter, R. 1978. On closed world data bases. In H. Gallaire and J. Minker. Editors. *Logic and Data Bases*, pp. 55–76. New York: Plenum Press, 1978. Reprinted in (Ginsberg 1987), pp. 300–333.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proc. 4th International Joint Conference on Artificial Intelligence (IJCAI)*, Tbilisi, Georgia, USSR, pp. 206–14. Reprinted in (Allen, Hendler, & Tate 1990), pp. 162–70.

Sussman, G. J. 1974. The virtuous nature of bugs. In *Proc. Artificial Intelligence and the Simulation of Behaviour (AISB) Summer Conference*. Reprinted in (Allen, Hendler, & Tate 1990), pp. 111–17.

Thomason, R. H. 1974. Editor. *Formal Philosophy: Selected papers of Richard Montague*. New Haven: Yale University Press.

Turner, R. 1987. A theory of properties. In *Journal of Symbolic Logic 52(2):455–472.*

Wallen, L. 1987. Matrix proofs for modal logics. *Proc. 10th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 917–23.

Whitehead, A. N., and Russell, B. 1910. *Principia mathematica*. Cambridge: Cambridge University Press.