Open World Planning in the Situation Calculus

Alberto Finzi and **Fiora Pirri**
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
{alberto,fiora}@assi.dis.uniroma1.it

Ray Reiter
Department of Computer Science
University of Toronto
Toronto, Canada, M5S 1A4
reiter@cs.toronto.edu

Abstract

We describe a forward reasoning planner for open worlds that uses domain specific information for pruning its search space, as suggested by (Bacchus & Kabanza 1996; 2000). The planner is written in the situation calculus-based programming language GOLOG, and it uses a situation calculus axiomatization of the application domain. Given a sentence σ to prove, the planner regresses it to an equivalent sentence σ_0 about the initial situation, then invokes a theorem prover to determine whether the initial database entails σ_0 and hence σ. We describe two approaches to this theorem proving task, one based on compiling the initial database to prime implicate form, the other based on Relsat, a Davis/Putnam-based procedure. Finally, we report on our experiments with open world planning based on both these approaches to the theorem proving task.

Introduction

Currently, virtually all implemented deterministic planning systems make a closed world assumption that complete information is available about the initial state of the application domain. Conformant Graphplan (Smith & Weld 1998), CMBP (Cimatti & Roveri 1999) and the planner of (Rintanen 1999) are exceptions to this. So also are a few conditional planners incorporating "information gathering" actions, used to fill gaps in the planners' incomplete knowledge base (e.g. (Golden, Etzioni, & Weld 1994; de Giacomo *et al.* 1997)). Open worlds preclude direct appeal to most planning algorithms in the literature, including the successful SAT (Kautz & Selman 1996) and Graphplan (Blum & Furst 1997) approaches.

In this paper, we describe the theoretical foundations and implementation for an open world planner *without* sensing actions; its job is to find straight line plans using only what is known about the (incomplete) initial state, together with general domain specific facts like state constraints and action preconditions and effects. Ours is a forward reasoning planner, using domain dependent information to prune its search space, as suggested by (Bacchus & Kabanza 1996; 2000); it is axiomatized entirely in the situation calculus.

The Situation Calculus and GOLOG

The situation calculus (McCarthy 1963) is a first order language for axiomatizing dynamic worlds. In recent years, it has been considerably extended beyond the "classical" language to include concurrency, continuous time, etc, but in all cases, its basic ingredients consist of *actions*, *situations* and *fluents*.

Actions Actions are first order terms consisting of an action function symbol and its arguments. In the blocks world, the action of moving block x onto block y might be denoted by the action term $move(x, y)$.

Situations A *situation* is a first order term denoting a sequence of actions. These sequences are represented using a binary function symbol do: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. So $do(\alpha, s)$ is like LISP's $cons(\alpha, s)$, or Prolog's $[\alpha \mid s]$. The special constant S_0 denotes the *initial situation*, namely the empty action sequence, so S_0 is like LISP's ( ) or Prolog's [ ]. Therefore, in a blocks world, the situation term

$$do(move(A, B), do(moveToTable(B), do(move(C, D), S_0)))$$

denotes the sequence of actions

$$[move(C, D), moveToTable(B), move(A, B)].$$

Notice that the action sequence is obtained from a situation term by reading the term from right to left.

Foundational axioms for situations are given in (Pirri & Reiter 1999).

Fluents Relations whose truth values vary from state to state are called *fluents*, and are denoted by predicate symbols with last argument a situation term. For example, in the blocks world, $on(x, y, s)$ might be a relational fluent, meaning that in that state of the world reached by performing the action sequence s, block x will be on block y.

Axiomatizing a Domain Theory

A domain theory is axiomatized in the situation calculus with four classes of axioms (More details in (Pirri & Reiter 1999)):

1. **Action precondition axioms**. There is one for each action function $A(\vec{x})$, with syntactic form

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s).$$

Here, $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. These characterize the preconditions of the action A.

2. **Successor state axioms**. There is one for each fluent $F(\vec{x}, s)$, with syntactic form
$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s),$$
where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $a, s, \vec{x}$. These characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of the current situation s, and they embody a solution to the frame problem for deterministic actions (Reiter 1991).
3. **Unique names axioms for actions**. These state that the actions of the domain are pairwise unequal.
4. **Initial database**. This is a set of first order sentences whose only situation term is S_0 and it specifies the initial state of the domain.

Example 1 The following are successor state and action precondition axioms for a blocks world used in the implementation described below.

Action Precondition Axioms

$$Poss(move(x, y), s) \equiv clear(x, s) \wedge clear(y, s) \wedge x \neq y,$$

$$Poss(moveToTable(x), s) \equiv clear(x, s) \wedge \neg onTable(x, s).$$

Successor State Axioms

$$clear(x, do(a, s)) \equiv (\exists y)\{[(\exists z)a = move(y, z) \vee a = moveToTable(y)] \wedge on(y, x, s)\} \vee clear(x, s) \wedge \neg(\exists y)a = move(y, x),$$

$$on(x, y, do(a, s)) \equiv a = move(x, y) \vee on(x, y, s) \wedge a \neq moveToTable(x) \wedge \neg(\exists z)a = move(x, z),$$

$$onTable(x, do(a, s)) \equiv a = moveToTable(x) \vee onTable(x, s) \wedge \neg(\exists y)a = move(x, y).$$

Planning in the Situation Calculus

The classical definition of planning is (Green 1969).

Definition 1 Plans
Let $\mathcal{D}$ be a background situation calculus axiomatization for some domain, and $G(s)$ a situation calculus formula – the *goal* – with one free situation variable s. A situation term $do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots))$ that mentions no free variables is a *plan for G* iff

$$\mathcal{D} \models executable(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots))) \wedge G(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots))).$$

Here, the *executable* expression is an abbreviation for $Poss(\alpha_1, S_0) \wedge Poss(\alpha_2, do(\alpha_1, S_0)) \wedge \cdots \wedge Poss(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots))$. So on this definition, planning is a theorem-proving task:
Determine a sequence $\alpha_1, \ldots, \alpha_n$ of variable free action terms such that $G(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots)))$ is provable from the background axioms $\mathcal{D}$, and moreover, this action sequence is executable, meaning that each of its action's preconditions are provable in that situation in which it is to be performed. This is the formal foundation for our open world planner.

GOLOG

Our planner is implemented in the situation calculus-based programming language GOLOG (Levesque *et al.* 1997), a language for defining complex actions in terms of a set of primitive actions axiomatized, as described above, in the situation calculus. It has the standard – and some not so standard – control structures found in most Algol-like languages.

1. *Sequence:* α ; β. Do action α, followed by action β.
2. *Test actions:* p? Test the truth value of expression p in the current situation.
3. *Nondeterministic action choice:* $\alpha \mid \beta$. Do α or β.
4. *Nondeterministic choice of action arguments:* $(\pi\, x)\alpha$. Nondeterministically pick a value for x, and for that value of x, do the action α.
5. *Procedures, including recursion.*

The semantics of GOLOG programs is defined (see (Levesque *et al.* 1997)) by macro-expansion, using a ternary relation *Do*. $Do(program, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that s' is one of the situations reached by evaluating the GOLOG *program*, beginning in situation s. Therefore, to execute *program*, one *proves*, using the situation calculus axiomatization of some background domain (e.g. the axioms of Example 1), the situation calculus formula $(\exists s)Do(program, S_0, s)$. Any binding for s obtained by a constructive proof of this sentence is an execution trace, in terms of the primitive actions, of the *program*. A GOLOG interpreter, written in Prolog, is described in (Levesque *et al.* 1997); this is the implementation used for our planner.

A Depth-First Forward Planner

We can now present our planner, written in GOLOG.

proc $wspdf(n)$
$goal? \mid [n > 0? \,;\, (\pi\, a)(primitive\_action(a)? \,;\, a) \,;\, \neg badSituation? \,;\, wspdf(n-1)]$
endProc

$wspdf(n)$[1] expects the user to provide n, a depth bound, *goal*, a planning goal, *primitive_action*, a predicate characterizing what are the primitive actions of the domain, and finally, an axiomatization of a domain dependent predicate *badSituation*, used to filter out partial plans that are known to be fruitless.

Like any GOLOG program, the planner is executed by proving $(\exists s)Do(wspdf(n), S_0, s)$. Therefore, we start with S_0 as the current situation. In general, if σ is the current situation, $wspdf(n)$ succeeds if $goal(\sigma)$ can be proved, or if $n > 0$ and a primitive action a can be chosen (nondeterministically) such that $Poss(a, \sigma)$ is provable and such that after "performing" a, meaning that $do(a, \sigma)$ becomes the new current situation, $\neg badSituation(do(a, \sigma))$ can be proved, and $wspdf(n-1)$ succeeds with $do(a, \sigma)$ as the current situation. On success, the current situation is a plan for the *goal*. Therefore, the planner works depth-first, generating subplans by increasing length, filtering these with the *badSituation* predicate, and testing the survivors

[1] *wspdf* stands for the World's Simplest Depth First Planner

against the *goal*. The *badSituation* filter is our version of Bacchus and Kabanza's use of domain specific information for pruning the search tree of bottom-up planners (Bacchus & Kabanza 2000).

A Regression-Based Theorem Prover

The planner *wspdf* generates potential plans of the form $do(\alpha_n, \dots, do(\alpha_1, S_0)\cdots)$, where the α_i are action terms; then it tests these potential plans against *goal* and *badSituation*. So the test expressions that must be proved are of the form $W(do(\alpha_n, \dots, do(\alpha_1, S_0)\cdots))$, for formulas $W(s)$ with a single free situation variable s. These are the kinds of sentences for which regression was designed (Pirri & Reiter 1999). Essentially, regression uses successor state axioms to replace a sentence of the form $W(do(\alpha_n, \dots, do(\alpha_1, S_0)\cdots))$ by a logically equivalent sentence *about the initial situation only*, and the original sentence is provable iff the regressed sentence is provable *using only the initial database together with the unique names axioms for actions*. So our strategy will be this:

1. Eliminate the quantifiers in the sentence $W(do(\alpha_n, \dots, do(\alpha_1, S_0)\cdots))$. Here, we make the domain closure assumption that the only function symbols for domain objects are constants, and there are just finitely many individuals in the domain of discourse. Moreover, we allow only *typed quantification*. Formally, a *type* $\tau(x)$ is an abbreviation for a description of a finite domain of constants:
$$\tau(x) \stackrel{def}{=} x = T_1 \vee \cdots \vee x = T_k,$$
where $T_1, \dots, T_k$ are all constants. Now, introduce *typed quantifiers* $(\forall x : \tau)$ and $(\exists x : \tau)$ according to:
$$(\forall x : \tau)\phi(x) \stackrel{def}{=} (\forall x).\tau(x) \supset \phi(x),$$
$$(\exists x : \tau)\phi(x) \stackrel{def}{=} (\exists x).\tau(x) \wedge \phi(x).$$
Then typed quantification of formulas can be reduced to conjunctions and disjunctions according to the following equivalences
$$(\forall x : \tau).\phi(x) \equiv \phi(T_1) \wedge \cdots \wedge \phi(T_k),$$
$$(\exists x : \tau).\phi(x) \equiv \phi(T_1) \vee \cdots \vee \phi(T_k).$$
Therefore, quantifier elimination for the sentence $W(do(\alpha_n, \dots, do(\alpha_1, S_0)\cdots))$ amounts to replacing W's quantified subformulas by conjunctions and disjunctions according to these equivalences.
2. Regress the resulting sentence to a sentence about the initial situation only.
3. Convert the regressed sentence to clausal form.
4. Determine whether all clauses of this clausal form are entailed by the initial database. If so, report QED; else report failure.

This is what our implementation does, with one important difference: Rather than regress the entire sentence to the initial situation, it does a depth first regression of the components of the sentence, hoping that the regressed component will simplify in such a way that the remaining components need not be regressed. For example, suppose, in regressing $P \wedge Q$ we first regress P to get R. If R simplifies to $false$, there is no point in next regressing Q, since in any event, the regressed form of $P \wedge Q$ will be $false$. There is a dual principle for regressing formulas of the form $P \vee Q$.

Regression requires successor state axioms, and we allow for these by user-provided Prolog assertions of the form `Atom <=> Expression`. Now we can describe the final details of the regression theorem prover. In regressing an atom `A`, there are two possibilities:

1. `A` has a definition of the form `A <=> W`. This means that `A` has a successor state axiom or is a defined atom, and to regress `A`, we need to regress `W`.
2. `A` has no definition of the form `A <=> W`. Therefore, either `A` is not a fluent, or it is, but its situation argument is `s0`, so the regression is finished.

The following are the top-level Prolog clauses in our implementation for the regression theorem-prover `prove`, as just described:

A Regression Theorem Prover

```
prove(W):- eliminateQuantifiers(W,I),
           simplify(I,Simp), regress(Simp,R),
           clausalForm(R,Clauses),
           databaseEntails(Clauses).

regress(P & Q, R) :- regress(P,R1),
        (R1 = false, R = false, ! ;
         regress(Q,R2), simplify(R1 & R2,R)).
regress(P v Q, R) :- regress(P,R1),
        (R1 = true, R = true, ! ;
         regress(Q,R2), simplify(R1 v R2,R)).
regress(-P,R) :- regress(P,R1),
                 simplify(-R1,R).
regress(A,R) :- isAtom(A), A <=> W,
  /* A is a defined atom.
     Retrieve and regress its definition. */
                eliminateQuantifiers(W,I),
                simplify(I,S), regress(S,R).
regress(A,R) :- isAtom(A), not A <=> W,
 /* A is an atom, but it has no definition,
    so the regression is finished. */
         (A = false, R = false, ! ; R = A).
```

Here, `databaseEntails(Clauses)` is a call to a theorem prover for determining whether `Clauses` are a logical consequence of the clauses for the initial database. The precise details of this theorem prover are still open; to complete the implementation, this must be specified, and that is the topic of the next section.

Theorem Proving in the Initial Database

For open world planning, our task is to implement a theorem prover for determining whether `databaseEntails(Clauses)`, as required by the regression-based theorem prover. Here, `Clauses` are the clauses obtained by regressing the expression to be proved. Our overriding concern must be to make this theorem proving task as efficient as possible because the prover will be called each time a test expression must be evaluated, and in the process of searching for a plan, such test expressions (e.g. $on(a, b, do(move(c, d), do(moveToTable(d), S_0))))$ are generated and tested a huge number of times. We have experimented with two approaches, one based on compiling the initial database to prime implicate form, the other using an on-line theorem prover.

Prime Implicates and Compiling an Initial Database

We begin with the initial database, which, being open world, can be any sentences about S_0. Our approach is to transform these sentences into their logically equivalent *prime implicates*.

Definition 2 Prime Implicate
Let $\mathcal{K}$ be a set of clauses. A clause C is a *prime implicate* of $\mathcal{K}$ iff C is not a tautology, $\mathcal{K} \models C$, and there is no clause $C' \neq C$ such that C' subsumes C and $\mathcal{K} \models C'$.[2]

Theorem 1 *(Quine 1959) Suppose $\mathcal{K}$ is a set of clauses, and $pi(\mathcal{K})$ is the set of all of $\mathcal{K}$'s prime implicates. Then $\mathcal{K}$ and $pi(\mathcal{K})$ are logically equivalent. Moreover, for any non-tautologous clause C, $\mathcal{K} \models C$ iff there is a clause $\Pi \in pi(\mathcal{K})$ such that Π subsumes C.*

This tells us, first, that we can safely replace $\mathcal{K}$ by its prime implicates, and secondly, with these equivalent clauses in hand, we can quickly determine whether a given clause is entailed by $\mathcal{K}$. So it seems that we need only to compute $\mathcal{K}$'s prime implicates, and thereafter we have efficient theorem proving that can be performed in time linear in the number of prime implicates. These prime implicates act like a compiled form of $\mathcal{K}$: All the "hard" reasoning is done at compile time, in computing the prime implicates; after that, reasoning becomes easy. Of course, there is no free lunch, so we have to expect that the compilation phase will have high complexity, and indeed, this is so. In the worst case, the number of prime implicates of a set of clauses is exponential in the number of distinct atoms in those clauses.

The first stage of our implementation converts the initial database to clausal form. In doing so, it first eliminates quantifiers in the sentences of the initial database, making use of various logical simplifications, for example, replacing `X = X` by `true`, `-true` by `false`, `P & true` by `P`, etc. Finally, the implementation computes the prime implicates of these clauses, using a straightforward algorithm based on (Quine 1959).

On-Line Theorem Proving

The alternative to compiling the initial database is on-line theorem proving. This involves a once-only conversion of the initial database to clausal form by quantifier elimination as described in the previous section, but without further processing these clauses into prime implicates. Subsequently, whenever it is required to establish `databaseEntails(Clauses)` for a regressed set of `Clauses`, a clausal form theorem prover is invoked. The advantage of this on-line approach is that it avoids the expensive prime implicate computation. The disadvantage is that the theorem proving task becomes much more expensive.

An Open Blocks World

We illustrate an open world axiomatization for the blocks world. Recall that $wspdf(n)$ searches for plans bottom-up, filtering out useless subplans with a user supplied, domain specific *badSituation* predicate. We next describe the predicate we used for our blocks world implementation.

Let $goodTower(x, s)$ be true whenever, in situation s, x is a good tower, meaning that x is the top block of a tower of blocks that is a sub-tower of one of the goal towers. We suppose the planner has available to it a description of all the good towers corresponding to its planning goal. The following are some natural *badSituation*s:

1. The situation resulting from moving a block off a good tower.
2. The situation resulting from moving a block onto a good tower, if the resulting tower is a bad tower.
3. *Opportunistic rule:* The situation resulting from creating a bad tower by moving a block to the table, and some other action could have been performed instead that creates a good tower.

badSituation also imposes certain canonical ordering rules on plans, which we do not describe here. The following Prolog clauses implement rules 1 and 3:

Some Bad Situations for a Blocks World

```
badSituation(do(move(X,Y),S)) :-
        prove(-goodTower(X,do(move(X,Y),S))).
badSituation(do(moveToTable(X),S)) :-
   prove(-goodTower(X,do(moveToTable(X),S))),
   existsActionThatCreatesGoodTower(S).
existsActionThatCreatesGoodTower(S) :-
      (A = move(Y,X) ; A = moveToTable(Y)),
      poss(A,S), prove(goodTower(Y,do(A,S))).
```

Next, we present an example open world blocks problem with 12 blocks, arranged as indicated in the figure.

A Blocks World Problem with Incomplete Initial Situation

```
/* Initial situation: Only the blocks so
   indicated have been specified to be clear.

                        d      m <-- clear;
    clear --> p         a      not on table
    not on--> n        /\
    table             ?/  \?      f <-- not
                      /    \       on table
              g       b     \
              h       c      e     k
           --------------------------

Goal situation        d     k
                      h     g
                      b     m
                      e     f
                      a     c
                   -----------             */

goal(S) :- prove(on(d,h,S) & on(h,b,S) & on(b,e,S) &
  on(e,a,S) & ontable(a,S) & on(k,g,S) & on(g,m,S) &
  on(m,f,S) & on(f,c,S) & ontable(c,S) ).

goodTower(X,S) <=> X = a & ontable(a,S) v
  X = e & on(e,a,S) & ontable(a,S) v
  X = b & on(b,e,S) & on(e,a,S) & ontable(a,S) v
  X = h & on(h,b,S) & on(b,e,S) & on(e,a,S) &
          ontable(a,S) v
  X = d & on(d,h,S) & on(h,b,S) & on(b,e,S) &
          on(e,a,S) & ontable(a,S) v
  X = c & ontable(c,S) v
  X = f & on(f,c,S) & ontable(c,S) v
```

[2] A clause is a *tautology* iff it contains A and $\neg A$ for some atom A. Clause C *subsumes* clause C' iff each literal of C occurs in C'.

```
  X = m & on(m,f,S) & on(f,c,S) & ontable(c,S) v
  X = g & on(g,m,S) & on(m,f,S) & on(f,c,S) &
          ontable(c,S) v
  X = k & on(k,g,S) & on(g,m,S) & on(m,f,S) &
          on(f,c,S) & ontable(c,S).

/* Initial database. All references to clear and
   ontable have been eliminated, via their
   definitions, in favour of on.  */

axiom(all([y,block],-on(y,m,s0))). % m is clear.
axiom(all([y,block],-on(y,p,s0))). % p is clear.
axiom(all([y,block],-on(k,y,s0))). % k on the table.
axiom(all([y,block],-on(c,y,s0))). % c on the table.
axiom(all([y,block],-on(e,y,s0))). % e on the table.
axiom(all([y,block],-on(h,y,s0))). % h on the table.
axiom(on(b,c,s0)).    axiom(on(d,a,s0)).
axiom(on(g,h,s0)).    axiom(on(p,n,s0)).
axiom(on(a,b,s0) v on(a,e,s0)). % a is on b or on e;
axiom(all([x,block],on(x,b,s0) => x = a)). % nothing
axiom(all([x,block],on(x,e,s0) => x = a)). % else
                                       % is on b or e.
axiom(some([x,block],on(f,x,s0))). % f not on table.
axiom(some([x,block],on(m,x,s0))). % m not on table.
axiom(some([x,block],on(n,x,s0))). % n not on table.

%  Initial state constraints.

axiom(all([x,block],all([y,block],
                      on(x,y,s0) => -on(y,x,s0)))).
axiom(all([x,block],all([y,block], all([z,block],
              on(y,x,s0) & on(z,x,s0) => y = z)))).
axiom(all([x,block],all([y,block], all([z,block],
              on(x,y,s0) & on(x,z,s0) => y = z)))).

% clear & ontable defined in the initial situation.

clear(X,s0) <=> all([y,block],-on(y,X,s0)).
ontable(X,s0) <=> all([y,block],-on(X,y,s0)).

%  Domain of blocks.

domain(block,[a,b,c,d,e,f,g,h,k,m,n,p]).

%  Action preconditions.

poss(move(X,Y),S) :-
      findall(Z,(domain(D), member(Z,D),
                 prove(clear(Z,S))),L),
      member(X,L), member(Y,L), not X = Y.
poss(moveToTable(X),S) :- domain(D), member(X,D),
               prove(clear(X,S) & -ontable(X,S)).

/* Successor state axioms  */

clear(X,do(move(U,V),S)) <=>
                 on(U,X,S) v -(X = V) & clear(X,S).
clear(X,do(moveToTable(U),S)) <=>
                 on(U,X,S) v clear(X,S).
on(X,Y,do(move(U,V),S)) <=>
             X = U & Y = V v -(X = U) & on(X,Y,S).
on(X,Y,do(moveToTable(U),S)) <=>
                         -(X = U) & on(X,Y,S).
ontable(X,do(move(U,V),S)) <=>
                      -(X = U) & ontable(X,S).
ontable(X,do(moveToTable(U),S)) <=>
                         X = U v ontable(X,S).

primitive_action(move(X,Y)).
primitive_action(moveToTable(X)).
```

There are four things to note about these axioms:

1. The initial database is defined only using the fluent *on*, and not *clear* and *ontable*. In the blocks world, the fluent *on* is primitive, and fluents *clear* and *ontable* can be defined in terms of it:

   $$clear(x,s) \equiv (\forall y)\neg on(y,x,s),$$
   $$ontable(x,s) \equiv (\forall y)\neg on(x,y,s).$$

   Thus, instead of representing the initial fact that h is on the table by $ontable(h, S_0)$, we elected instead to use $(\forall y)\neg on(h,y,S_0)$; similarly for *clear*. With this choice, the initial database does not include facts about *clear* and *ontable*. This considerably reduces the number of clauses for the initial database because these will not include redundant clauses involving *clear* and *ontable*. This, in turn, considerably improves the theorem proving efficiency.
2. The initial database axioms include three *state constraints*, relativized to the initial situation. The *general* state constraints, e.g. $(\forall x,y,s).on(x,y,s) \supset \neg on(y,x,s)$, are not among our axioms because, except for the case $s = S_0$, they are entailed by the successor state and action precondition axioms. See (Lin & Reiter 1994) for a discussion of this issue. In general, for open world planning, the initial database must include all state constraints for the application domain, relativized to the initial situation.
3. The clauses for `poss` and `goal` assume responsibility for calling `prove` on appropriate formulas.
4. The successor state axioms differ from Example 1, which universally quantify over all actions a. In contrast, the open world blocks world axiomatization uses two clauses for `clear`, one for action `move(U,V)`, the other for action `moveToTable(U)`. There is no deep reason for this choice. It was made only to simplify the implementation of the simplification routine `simplify` used by the regression theorem prover. To see why, consider the successor state axiom for *clear* in example 1, and consider an instance $move(u,v)$ of a in this sentence:

   $$\begin{aligned} &clear(x, do(move(u,v),s)) \equiv \\ &\quad (\exists y)\{[(\exists z)move(u,v) = move(y,z) \vee \\ &\qquad move(u,v) = moveToTable(y)] \wedge on(y,x,s)\} \vee \\ &\quad clear(x,s) \wedge \neg(\exists y)move(u,v) = move(y,x). \end{aligned}$$

   Using the unique names axioms for actions, and some elementary logic, this can be simplified to yield the logical form of the first successor state axiom for `clear` in the above blocks world axiomatization. Obtaining this logical form from the general successor state axiom was straightforward, but required a lot of simplification based on reasoning about quantifiers, equality and unique names axioms for actions. To avoid having to implement such simplification routines we have opted instead for the user of the system to do this herself, in advance, and to represent the results of these simplifications directly by successor state axioms, particularized to each action, as was done in the above axiomatization.

Experimental Results

We tested our planner on two classes of problems: One class consists of variants of the blocks world problem of the previous section in which we vary the extent to which individual blocks have known information associated with them. The other problem class was drawn from the logistics domain, and again, we varied the amount of known information about the domain's individuals. Moreover, we experimented with two versions of the planner's theorem prover: the prime implicate approach described above, and an on-line theorem prover Relsat 1.0, an implementation of the Davis-Putnam algorithm based on (Bayardo & Schrag 1997).

In designing our experiments, we introduced the concept of an *unknown* for the purposes of measuring a problem's degree of incompleteness. For a given domain individual, an unknown is any property of that individual that is not entailed by the initial database, and whose negation also is not entailed. For the blocks world problem of the previous section, $clear(d)$ is an unknown property of d, because the axioms for the initial database entail neither it, nor $\neg clear(d)$. $(\lambda x)on(m, x)$ is an unknown property of m, and $(\lambda x)(x = b \vee x = e) \wedge on(a, x)$ is an unknown property of a. In total, there are 8 unknown properties for this example.

The Blocks World

Here, we used the axiomatization of the blocks world given in the previous section, but varied the initial situation (so only the first problem in the set of experiments corresponds to the picture associated with those axioms). The remaining four blocks world problem instances used in our experiments were obtained from the problem given in the previous section by adding more blocks to it, and by varying the amount of known information about these additional blocks. The tables given below summarize the comparative performances of Relsat vs the prime implicate implementations of the theorem proving component of our open world planner.[3]

Relsat					
Blocks	12	20	20	22	22
Unknowns	8	8	10	14	10
Plan (sec)	19.7	19.0	31.2	38.7	38.2
Clauses	175	449	502	624	586
Compile (sec)	5.1	44.8	50.1	79.6	82.1
Plan length	14	14	17	17	17

Prime Implicates					
Blocks	12	20	20	22	22
Unknowns	8	8	10	14	10
Plan (sec)	3.9	19.8	-	-	-
Clauses	175	449	502	624	586
Implicates	248	569	-	-	-
Compile (sec)	15.7	119.6	-	-	-
Plan length	14	14	-	-	-

The Logistics Domain

Space limitations prevent us from presenting our situation calculus axiomatization for this domain. They are available, on request, from the authors. To generate open world problems for this domain, we selected a number of the standard closed world benchmarks, and "opened them up" by removing information from their initial state descriptions. As for the blocks world, our measure of the degree of incompleteness of these problems was defined by the number of unknown properties of all domain individuals. In the tables below, the problem numbers are those of the closed world instances of logistics problems used in the First International Planning Competition, described at www.informatik.uni-freiburg.de/~koehler/ipp.html. Our problems were obtained by "opening up" these problem instances.

Relsat: Logistics				
Problem	03	03	04	05
Unknowns	0	1	2	11
Plan (sec)	24.3	31.0	342.8	1405.0
Clauses	776	811	6772	9359
Compile (sec)	9.8	10.5	1146.6	1528.9
Plan length	9	10	11	19

Prime Implicate: Logistics				
Problem	03	03	04	05
Unknowns	0	1	2	11
Plan (sec)	19.0	24.8	262.5	stack o'flow
Clauses	776	811	6772	9359
Implicates	776	811	6772	9359
Compile (sec)	15.0	16.3	1178.3	2339.2
Plan length	9	10	11	-

Discussion

Our experiments suggest that open world planning is feasible for moderate sized problems (up to 17 step blocks world plans, and 19 step logistics plans). To our surprise, theorem proving with Relsat 1.0 was the clear winner over the prime implicate compilation approach. With larger problems, the latter was overwhelmed by the prime implicate computation, whereas Relsat found plans for all our problem instances. Perhaps a more sophisticated prime implicate algorithm would have helped here, e.g. (de Kleer 1992). We leave such considerations for the future.

Conformant Graphplan (Smith & Weld 1998), CMBP (Cimatti & Roveri 1999) and the planner of (Rintanen 1999) are the only other open world planners that we know. Cimatti and Roveri have extensively tested their planner against those of Smith/Weld and Rintanen, and their data suggests the superiority of their approach over these other planners. The examples on which we tested our planner did not intersect those of Cimatti and Roveri – we did not learn of their work until after we had conducted our experiments. However, we have since done some unsystematic runs with our planner on the bomb and the toilets problem that figured prominently in their experiments. This problem has some straightforward and natural *badSituation*s:

1. Flushing a toilet twice without an intervening package dunking creates a *badSituation*.
2. The problem has a high degree of symmetry, which can be substantially reduced by canonically ordering the packages and toilets:
   (a) Dunking a package creates a *badSituation* if there is an undunked package lower in the package ordering.
   (b) Dunking into a toilet creates a *badSituation* if there is an unclogged toilet lower in the toilet ordering.

[3] All CPU times here are for a SUN Sparc 10 Ultra, with 333 MHZ processor and 256 MB of RAM.

We tested our prime implicate-based planner, using these *badSituations*, on several of the Cimatti and Roveri test problems. None caused it any difficulties. The smallest problem their planner could not solve was BMTC(10,3) – 10 packages, 3 toilets – with high uncertainty, meaning that initially it is unknown whether any of the toilets are clogged; our planner found a 20 step plan in 0.32 seconds. The biggest problem we ran was BMTC(40,6) – 40 packages, 6 toilets – with high uncertainty, producing an 80 step plan in 114 seconds.

In all fairness to CMBP, we must emphasize that it is a domain independent planner, whereas ours relies heavily on its problem specific *badSituations*. Moreover, CMBP returns all minimal length plans, while ours returns plans one at a time, and these need not be minimal length. Finally, CMBP provides for actions with nondeterministic effects; for us, all actions must be deterministic. On the other hand, we can't think of any reasons for not exploiting domain specific information in planning when it is so obviously useful.

Our planner differs considerably from those of Bacchus/Kabanza in its theoretical foundations and in its implementation. Theoretically, it is based entirely on the situation calculus and Green's Definition 1 for planning, and is arguably more "logically pure" and transparent than the latter, which rely on a combination of STRIPS-like operators and a linear temporal logic. In its implementation, our planner differs in one fundamental respect. The Bacchus/Kabanza planners maintain a current database by *progressing* the previous database in response to the last planned action, whereas our planner does no database progression; instead, it maintains only the initial database, and computes entailments using goal regression together with theorem-proving relative to the initial database. There is a good reason for this: Except for one special case described in (Lin & Reiter 1997), there are no known provably correct and efficient algorithms for progressing an incomplete initial database. Despite these technical differences, our planner is very much in the spirit of the Bacchus/Kabanza approach, and our experiences reinforce their arguments in favor of exploiting domain specific control information in planning systems.

In addition to the open world planner described here, a variety of closed world, forward reasoning, situation calculus planners based on the ideas of Bacchus and Kabanza have also been implemented (Reiter 1999). These include planners for "classical" (totally ordered) problems, as well as for concurrency, and temporally ordered processes, as exemplified by a multi-handed blocks world agent.

(Levesque 1996) has elegantly generalized Green's account of planning in the situation calculus (Definition 1) to *conditional plans*, and we believe that the methods of this paper can be adapted to implement conditional planners based on his foundations.

References

Bacchus, F., and Kabanza, F. 1996. Planning for temporally extended goals. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'96)*, 1215–1222.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Bayardo, R., and Schrag, R. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'97)*, 203–208.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Cimatti, A., and Roveri, M. 1999. Conformant planning via model checking. In Biundo, S., ed., *Proc. ECP99: European Conference on Planning*. Springer-Verlag.

de Giacomo, G.; Iocchi, L.; Nardi, D.; and Rosati, R. 1997. Planning with sensing for a mobile robot. In *Preprints of the Fourth European Conf. on Planning*, 158–170.

de Kleer, J. 1992. An improved incremental algorithm for generating prime implicates. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'92)*, 780–785.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Efficient sensor management for planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'94)*, 1048–1054.

Green, C. 1969. Theorem proving by resolution as a basis for question-answering systems. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. New York: American Elsevier. 183–205.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'96)*, 1194–1201.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: a logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions* 31(1-3):59–83.

Levesque, H. 1996. What is planning in the presence of sensing? In *Proceedings of the National Conference on Artificial Intelligence (AAAI'96)*, 1139–1146.

Lin, F., and Reiter, R. 1994. State constraints revisited. *J. of Logic and Computation, special issue on actions and processes* 4:655–678.

Lin, F., and Reiter, R. 1997. How to progress a database. *Artificial Intelligence* 92:131–167.

McCarthy, J. 1963. Situations, actions and causal laws. Technical report, Stanford University. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

Pirri, F., and Reiter, R. 1999. Some contributions to the metatheory of the situation calculus. *Journal of the ACM* 46(3):261–325.

Quine, W. 1959. On cores and prime implicants of truth functions. *American Math. Monthly* 66:755–760.

Reiter, R. 1991. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. San Diego, CA: Academic Press. 359–380.

Reiter, R. 1999. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. In preparation. Draft available at http://www.cs.toronto.edu/~cogrobo/.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research* 10:323–352.

Smith, D., and Weld, D. 1998. Conformant graphplan. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'98)*, 889–896. AAAI Press/MIT Press.