A Declarative Implementation of Planning with Control*

Aarati Parmar
Gates Building, 2A wing
Stanford University, Stanford, CA 94305-9020, USA
aarati@cs.stanford.edu

Abstract

Being able to declaratively specify control within a planning (or theorem-proving) paradigm is necessary if we are to build intelligent machines. This paper introduces an implementation of planning using situation calculus, in the answer set programming paradigm. Our forward-chaining planner has control mechanisms which are based on three ideas: (1) controlling the domain of discourse, (2) *mental situations*, and (3) *contexts*. We recognize that the underlying computational mechanisms do not take advantage of these control mechanisms. In fact, our efficiency drops due to the addition of these mechanisms. However, we believe that a declarative specification will lead to an understanding, and creation of, better planning frameworks.

Introduction

Green style planning (Green 1969) works but is not practical, largely due to the explosion of the search space within the theorem prover. There is a lack of control in the paradigm. STRIPS (Fikes & Nilsson 1971) is the workable successor. STRIPS trades off some expressivity to be practical. In its framework, conjunctions of literals are manipulated, as opposed to first order sentences. STRIPS also lacks situations, which makes it impossible to compare two different situations within the framework. We cannot plan by comparing situations, as in the use of the $better(s, s')$ predicate in (Huberman 1968). More importantly, STRIPS contains no declarative formalism for control. We prefer a planning framework that has a clear, declarative mechanism for control, without giving up expressivity of plan operators.

Previous Approaches for Control in Planning

Systems such as SOAR, PRODIGY, and UCPOP use some sort of rule based control. Newer forward-chaining planners such as TLPLAN and TALPLAN use first order logic formulas, with some temporal information, to control selections of actions in the search. We focus on these two systems as they are most similar to our approach.

(Bacchus & Kabanza 2000)'s TLPLAN starts with an initial state, and looks for a sequence of ADL (Pednault 1989) actions which satisfies a linear temporal logic (LTL) (Emerson 1990) formula. A state is defined as a collection of literals, just as in STRIPS. LTL formulas include regular first order logic formulas, closed under the application of four temporal operators $\Box$, $\bigcirc$, $\diamond$, and U, and another modal operator GOAL. An LTL formula is evaluated with respect to a state. Formulae lacking temporal operators are evaluated entirely within the present state, in the obvious way. The four temporal operators, on the other hand, require future states for evaluation as well. For example, $\Box p$ holds at state s if p is true in every future state of s, while $\bigcirc p$ is true at s if p holds in the state immediately following s. $\text{GOAL}(p)$ is true if p holds in every one of the goal states (which are specified beforehand).

These temporal operators constrain sequences of states, based on the fluents that are true in them. Control is achieved by writing LTL formulas to constrain future states. For example in BlocksWorld, one could write (in their notation)

$$\Box(\forall[x : on(x, table)](\text{GOAL}(on(x, table)) \implies \bigcirc on(x, table)))$$

which means that for any future state, any block x currently on the table, that is supposed to be on the table in the goal state, must remain on the table in the succeeding state. This is one possible control rule that prevents unnecessary actions.

TALPLAN (Doherty & Kvarnström 1999) is inspired by TLPLAN. TALPLAN is based on a narrative-based linear temporal logic known as TAL (Doherty *et al.* 1998). TALPLAN forward chains just as TLPLAN does, except that instead of LTL, it uses TAL formulas, which explicitly refer to timepoints. TALPLAN is more expressive than TLPLAN, as actions that have durations, are concurrent, and have delayed effects can be represented. Amazingly enough, this extra expressivity does not hamper TALPLAN's performance. In fact it outperforms TLPLAN, both in terms of speed and memory.

Answer Set Planning

In answer set planning (ASP), a set of logic programming rules define both static and dynamic constraints on the domain. There is another set of rules which define the possible actions that may be taken at each time instant. Each stable model of this program, or *answer set*, contains a history

*This research has been partly supported by SNWSC contract N66001-00-C-8018.

of actions, which corresponds to a plan that obeys the constraints.

ASP finds its roots in (Subrahmanian & Zaniolo 1995). (Subrahmanian & Zaniolo 1995) provides a translation of STRIPS planning problems into logic programs such that stable models of the program correspond to a working plan. Building on this, (Dimopoulos, Nebel, & Koehler 1997) uses STRIPS operators, but allows parallel action execution. It also introduces some optimizations to the computation. This paper builds on the exposition given in (Lifschitz 2000). (Lifschitz 2000) is not restricted to STRIPS operators, but any action whose effects can be coded in a logic program. It also supports parallel action execution, and introduces the `GENERATE/DEFINE/TEST` paradigm for writing planning problems for logic programs.

Answer set planning is powerful for many reasons. For one, the domain can be declaratively specified, in a logic that is almost as expressive as first order logic, yet effectively computable. Secondly, with negation as failure, the representation of the domain can be more compact, and even more elaboration tolerant. For example, the [positive] common sense law of inertia may be encoded as: (Interpret "–" as classical negation, "`not`," negation as failure.)

```
holds(F,result(A,S) :-
      holds(F,S),
      not -holds(F, result(A,S)).
```

In other words, assume that a fluent `F` is true after the execution of an action if it held before and cannot be explicitly refuted. Thus we can add as many effect axioms as we like, and rely upon the above default to implement inertia.[1]

Answer set planning is powerful and efficient. But, each stable model corresponds to a different plan. There is no way to compare plans within the framework, or control the selection of actions in the search explicitly.[2] In this paper, we demonstrate a declarative approach to control in ASP. We use this paradigm because of its efficient computability, and expressivity, compared to Green style or STRIPS planning. Answer set planning may not immediately support all the types of actions TALPLAN can. But we believe some more interesting control mechanisms can be demonstrated in this framework.

In the next section we describe our approach to planning with control, in successively more detail, as we introduce the ideas of controlling the domain of discourse, *mental situations*, and *contexts*. We focus on BlocksWorld as our domain of choice. Then, we describe our experiences with other domains such as TireWorld and LogisticsWorld. Some comparisons are made with TLPLAN and TALPLAN. Finally, we conclude with some benefits and issues with our approach, along with some relations to first order logic theorem provers. We also explain how this declarative approach to planning could be implemented, so that it would be computationally efficient.

[1] It is often noted that this rule is semantically equivalent to (Reiter 1980)'s solution to the frame problem using default rules.

[2] Of course, even though in this work, we are declaratively specifying the computations, in the underlying mechanism (in this case LPARSE/SMODELS), the search is still out of our hands. We discuss this further in the Conclusions.

Approach

Our formalism differs slightly from that in (Lifschitz 2000). We use situations rather than timepoints. Beginning with facts about $s0$, we use our logic program rules to derive facts about situations of the form $result(a, s0)$. We then repeat the cycle on these new situations, searching until a situation which satisfies the goal conditions is found.

Currently, this loop is implemented by a perl script which forward chains our logic planning program on a different set of situations. The logic program itself has its own internal stages of "search" for each iteration. In the first stage, given some initial situations, the logic program applies domain constraints, and determines which actions are possible in those situations. `holds(prec(a),s)` is true if the action preconditions for `a` are met in `s`. Then in the second stage, the effect and inertial axioms, not to mention domain constraints, are applied to the situations `result(a,s)`. Once all the facts (of the form `holds(f,result(a,s)))` are derived about each situation, we can evaluate them to determine which to progress. In the last stage, we check to see if we have reached the goal. In reality, these computations are not being performed in this manner, but it is conceptually helpful to think of it as so.

Below we describe three special ideas that achieve control within this paradigm: (1) controlling the domain of discourse, (2) mental situations, and (3) contexts. Then we introduce our method for planning with control in more detail using these three concepts.

Controlling the Domain of Discourse

The principle is to control the domain of discourse over which the logic program runs, to minimize unnecessary grounding of rules. We take advantage of the workings of LPARSE (Syrjänen 2000) to accomplish this. LPARSE requires every rule on the logic program to be *strongly range restricted*. This means every variable in a rule must occur in a positive *domain predicate* in the body. A *domain predicate* is any predicate that is not defined recursively in terms of itself, even through other predicates. The objects over which any rule is grounded is limited to the extension of its domain predicates. Hence, controlling the extension of the domain predicates controls the entire program.[3]

Our planning program uses the domain predicate `sit(s)` to specify what objects are situations. In the first run of the planner, we only have `sit(s0)` and facts about `s0`, which is combined with the rest of the logic program

[3] LPARSE's strong range restriction is what forces us to have a loop paradigm in the first place! In order to have an answer set planning formulation in terms of situations, we would have to include all situations along the correct plan as terms *a priori*, since terms cannot be created on the fly. Since we do not know the solution to a plan before we solve it, we must include all situations, whose numbers grow exponentially in the length of the plan. The method in (Lifschitz 2000) addresses this problem by listing all legal timepoints (`time(0..maxtime)`).

describing the domain constraints and action effects. The resulting stable model of the program has facts only about situations of the form `result(a,s0)`. Our perl script extracts these situations from the stable model, labels them as `sit(result(a,s0))`, and applies them to our logic program again, along with the facts true in them. The next stable model has facts about situations of the form `result(a',result(a,s0))`, and so forth. The progression halts when a situation is reached which satisfies the goal condition. This halt condition is computed by the logic program, which signals a flag `gotgoal(s)`, where `s` is the situation satisfying the goal.

So far, this program as described only achieves a breadth first search over plans, without any optimizations. We better performance by only extracting `good` and `ok` situations for expansion. A `good` situation is informally defined as one which will definitely lead closer to the goal situation. An `ok` situation is one which will not lead [permanently] away from the goal.

`good` and `ok` are domain-independent notions. We can further refine these concepts given a particular domain. In BlocksWorld, for example, we use the concept of *final position* to define `good`. A block is in *final position* if it is currently on the table and should be in the goal state, or if it is on the block it should be on in the goal, and that block is in *final position*. Clearly one situation is better than another if it has strictly more blocks in *final position*. A situation is `good` if it has no situation better than it.

Only executing actions which lead to `good` situations is not enough to find a plan in BlocksWorld. As shown in Figure 1, sometimes there are no `good` situations. In this case, we use `ok` situations to break dependency cycles. In BlocksWorld, an `ok` situation is any resulting situation which is not `good` and where a block is moved to the table. This definition of `ok` will not lead "away" from the goal state, and will eventually lead to a `good` situation.

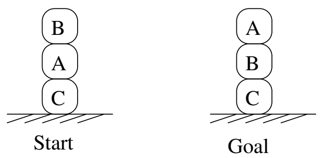

Figure 1: Only chaining on `good` situations is not enough to solve any instance of BlocksWorld.

Given these two descriptions of situations, we can easily formulate a strategy that forward chains only on `good` situations, and if they do not exist, uses `ok` situations. It should be obvious that this strategy will always lead to a plan in BlocksWorld. It also avoids problems like the Sussman anomaly.

Mental Situations

We can at least conceptually improve our search over plans even more, with the use of *mental situations* (Sierra 1998). While a situation is described as a snapshot of the world, a *mental situation* is a snapshot of the search for a plan over situations. This is easily accomplished by adding an additional mental situation argument to every literal in our logic program, so that it is only true (or visible) in certain *mental situations*. We can then envision *mental situations* as points of control within the search.

We can use mental situations to segment our search. The idea is sketched out graphically in Figure 2. Previously we described how our search begins with initial facts about a situation, and applies domain constraints to determine other properties of that situation in initial mental situation $m0$. In $m0$ we also find out which actions satisfy the `holds(prec(a),s)` predicate. Once all the facts are derived, we would like to filter facts of the form `holds(f,s)` and `holds(prec(a),s)` to a fresh new mental situation $m1$. In $m1$, we can apply effect and inertia axioms and domain constraints to those situations `result(a,s)` where `holds(prec(a),s)`.

Once we get all the facts about all the different resulting situations, we can apply control axioms to find out if any of them are `good` or `ok`. We can easily filter out only these situations, along with the facts true in them, to $m2$. In $m2$ we can also apply goal axioms to see if any of these situations satisfy the goal constraints.

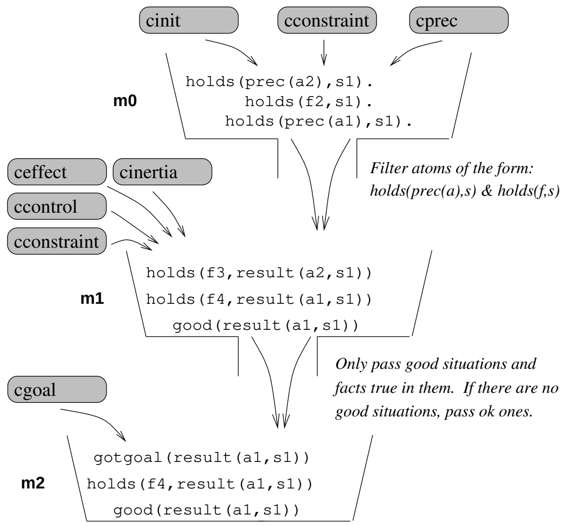

Figure 2: How mental situations work in our paradigm. The rounded off blocks correspond to sets of rules. The "vats" correspond to mental situations $m0$, $m1$, and $m2$ and can be envisioned as containers for computations of atoms by the rules.

The mindful reader will recognize that mental situations could be used to frame the entire search for a plan, within the one logic program, without having an outside loop. Candidate situations could be passed down a chain of mental situations, having their preconditions checked, progressed, and then classified as being `good`/`ok`. This computation could proceed until a goal is found. This elegant approach has two problems. The first, which was described earlier, is that due to the strong range restriction, every situation which might lie along the correct plan must be included *a priori* as a term in the program. We must also include all mental situations. The second is that adding mental situations multiplies the size of the program, which makes computation impossibly long in practice. This problem will be addressed further in the Conclusions.

Contexts

We can control what sets of propositions are visible in a mental situation. We should also be able to control which sets of rules are applicable in a mental situation. For example, when determining which actions are applicable in a situation, we only need the axioms that prove `holds(prec(a),s)`. To get the resulting situations, we only need effect, inertial, and domain constraint axioms. To determine which axioms are `good/ok`, we only need the control axioms. Finally, we only need a few special rules to determine whether a situation satisfies the goal conditions. Applying any other set of rules to the propositions visible in a mental situation is unnecessary, conceptually speaking. In each of these cases, there is a clearly defined set of rules which performs the required computation.

We can use *contexts* (McCarthy & Buvač 1998) to partition these rules into intuitive groupings. *Contexts* can be thought of as containers for sets of logical axioms. Thus we can have a *cprec* context, which contains all the rules which determine whether `holds(prec(a),s)`. Or a context $cgoal$, whose job is to determine which situations are in the goal state. By abstracting sets of rules, we can think about the results they entail more generally.

Now, we can further refine control by allowing only certain contexts, which correspond to sets of rules, to be visible at a mental situation. This means that we only want certain types of computations to be performed at a given mental situation, and no other.[4]

Planning in ASP, with Control

With these three tools in our box, we can declaratively specify all sorts of control mechanisms. Below in Figure 3 is a diagram of our planning paradigm.

The rounded blocks $cinit$, *cprec*, $cconstraint$, $cinertia$, $ceffect$, $ccontrol$, and $cgoal$ are contexts containing sets of rules. In a BlocksWorld formulation, $cinit$ would contain the literals:

```
holds(on(b,table), s0).
holds(on(a,b), s0).
holds(clear(a),s0).
```

cprec contains only one rule:

```
holds(prec(move(X,Y,Z)), S) :-
      holds(on(X,Y),S,M),
      holds(clear(X),S,M),
      holds(clear(Z),S,M),
      X!=Y, Y!=Z, X!=Z, X!=table,
      block(X;Y;Z), sit(S).
```

[4]Try expressing this sort of control in a first order resolution theorem prover!

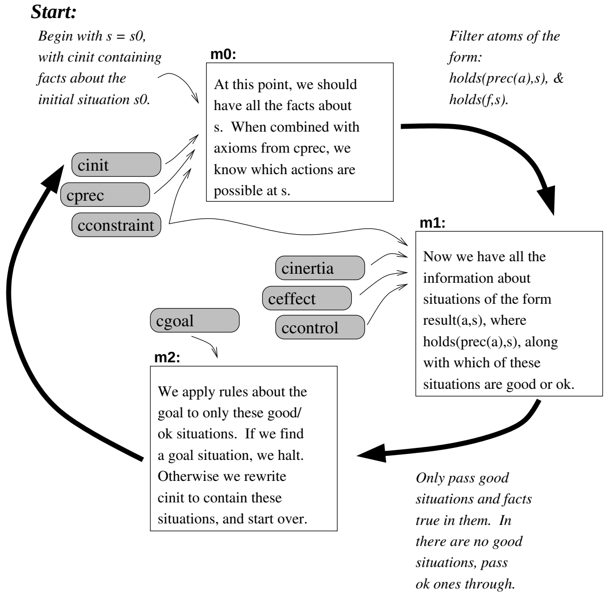

Figure 3: Diagram of our planning process. Each rounded block corresponds to a context. The boxes correspond to intermediate mental situations.

The planning process begins by having the only situation `sit(s0)`, with $cinit$ containing the initial facts, in mental situation $m0$. (Note that $cinit$ could contain multiple initial situations, allowing for parallel progression.) These facts combine in $m0$ with domain constraint rules ($cconstraint$) to produce the complete initial state of the world. These facts are also combined with the *cprec* rules, still in $m0$. Thus at $m0$ we should include the set of actions which are feasible in $s0$.

Then we "filter" facts about these complete situations down to $m1$, along with the permissible actions. In this mental situation, we apply effect ($ceffect$), inertial ($cinertia$), and domain constraint ($cconstraint$) axioms. At this point $m1$ should contain all the facts about each situation of the form `result(a,s0)`, where `holds(prec(a),s0)`. We only generate legal situations by adding the condition `holds(prec(a),s)` to the body of every effect and inertial axiom. Since logic programming does not support the contrapositive of implications, we only progress on *possible* actions, something that doesn't quite happen in Green style planning.

We can now examine and compare the resulting situations `result(a,s)`, to determine the best set of situations to pursue in the next progression. $ccontrol$ defines `good` and `ok` in terms of the planning domain. For example, in Blocksworld, these depend on the concept of *final position*, which is also defined. If there are no `good` situations, the `ok` situations are identified in $m1$. We pass these `good/ok` situations to $m2$, along with the facts true in them. In $m2$ we can check to see if we have reached the goal. If we have not, we replace the facts in $cinit$ with the `good/ok` situations, and call the entire loop again.

This simple strategy should solve any plan in situation calculus. We have expressed it entirely in the answer set paradigm, except for the outside process that (1) extracts out `good` and `ok` situations, along with the facts that hold in them, (2) adds them as legal situations, and (3) calls LPARSE/SMODELS on the new sets of rules.

In reality, we should stress again that these efforts only expand the search space, really making our computations only more inefficient.

Other Domains

We have computed and demonstrated our approach in the BlocksWorld domain, which is sure to reach a solution. Below are our experiences encoding other domains, and control strategies we used.

Tire World

We have formalized the `flat-tire-adl` domain written by Stuart Russell. This domain is more complicated than BlocksWorld. The goal of one problem is to change a tire, place all tools in the boot, and close the boot. The second goal is disjunctive, and in terms of the given predicates, the third goal is negative. However, we can still express these goals conjunctively with the use of extra logical machinery. We add the fluent `everytoolinboot` which is defined using negation as failure – `holds(neg(everytoolinboot),S)` if there is a tool *not* in the boot, and `holds(everytoolinboot,S)` if one cannot prove `holds(neg(everytoolinboot),S)`. (`neg` is axiomatized in the usual way.) This trick can also be used to express universal preconditions of actions. Conditional effects are also straightforward.

Our control rules are only limited by our imagination. For example, we only `cuss` when necessary, for otherwise this precondition-free action will clog the search. We can also detect whether the problem involves changing a tire (`tirechangeproblem`), which means we should get the wrench, jack, and pump ready for use. We could also label as "not `ok`" (`bad`) situations which contain two neighboring opposite actions, such as `result(loosen(N,H), result(tighten(N,H),S))`. In a domain such as TireWorld, with many undoable actions, "flip-flopping" with such sequences can further bog down the search.

Logistics World

In LogisticsWorld, packages are moved around cities by trucks, and between cities by airplanes. It is simple to write some simple control rules, such as loading a package into a truck if a package's goal position is somewhere else in the city, and then driving it to a location. A more elaborate hierarchy can be developed, classifying packages according to whether they only need to be moved somewhere else in the city, or to another city. We can optimize our UPS service by loading all necessary packages *before* driving the truck away. This is a disjunctive condition that is easily handled in logic programming, with the help of negation as failure.

Comparison with Other Approaches

The best planners with control to date (Bacchus 2000) include TLPLAN and TALPLAN. Computationally speaking they totally outperform our algorithm. We can only compare based on expressivity.

First of all, in TLPLAN and TALPLAN, control can only be framed in terms of *properties* of situations. Situations are essentially states. Situations cannot be indexed based on the actions which comprise it, for example. Situations also cannot be uniquely identified, and thus compared with each other. In our approach they are objects that can be manipulated, examined, and enumerated. Secondly, we can not only control our movement through the search space, but also how we extend it. By controlling the domain of discourse, we restrict the space we must explore. With the use of contexts, we can also restrict how we expand future situations. Third, we can define helpful intermediate fluents, such as `everytoolinboot`, or `tirechangeproblem`, in order to help determine the nature of our plan, and thus, what needs to be done next.

We do admit TALPLAN can handle a more expressive array of actions, including actions with duration, nondeterministic actions, delayed-effects of actions, and concurrent actions. Our sequential situation calculus framework forces us to chain actions. Solutions exist for adding almost all of the above goodies to sitcalc as in (Gelfond, Lifschitz, & Rabinov 1991), but these are more like patches glommed on top of the formalism rather than the essence of it.

Conclusions and Discussion

Benefits of this Approach

In this paper we have touted that mental situations, at least conceptually, eliminate unnecessary applications of rules, to inappropriate domain objects. Contexts not only organize sets of axioms into more intuitive groupings, they can also aid debugging a logic program. In a complicated planning domain, once we use negation of failure, all sorts of unintuitive dependencies can pop up. With the combination of mental situations and contexts, we can turn "off" and "on" different sets of rules to aid in debugging.

Since the planning formalization, along with the declarations of control, are entirely in logic, *almost any preference can be expressed*. At first it seems our paradigm is constrained by the `good/ok` restriction on visible situations. But, we can employ the more elegant treatment, where the outer loop is eliminated and planning happens along a chain of mental situations. Then, any set of facts that are logic-program-computable can be passed down to the next mental situation. Determining which facts are passed is also logic-program-computable.

Future Designs

This approach has not directly addressed or carefully handled deep problems in AI such as the ramification problem, incompleteness, or nondeterminism. In a non-monotonic framework, ramifications such as `prec`, must be handled gingerly. In general, they should not be subject to the laws of inertia. We have been roughly following the solution in (Lifschitz 1990), where we only apply inertia to some basic (*Frame*) fluents. We would like to be more systematic.

One possibility is to generate a mapping into our formalism from an action language such as C. We could also use CCALC, or PDDL, or some other action formalism/description that has been widely studied and understood. Then, given the mapping is sound, we could inherit whatever results that have already been proven.

We have also implicitly assumed that every situation can be described as a conjunction of fluents, possibly negated, and that the domain constraints, effect, and inertia axioms will determine the complete state of the world. We have not cleanly allowed for incomplete states, or nondeterministic actions, which could both create a disjunctive set of situations. In this answer set paradigm, this would correspond to multiple answer sets instead of one.

This problem of multiple answer sets, corresponding to different possible situations, could still be handled in this framework, with some extra work. Our paradigm would have to branch on every possible answer set. We could emulate *cautious* and *brave* plans by making sure plans work in all possible branchings, or only one. Nasty combinatorics problems would have to be addressed.

A more technical problem arises from our representation. The situation terms, being of the form $result(a_n, ..., result(a_1, s0)...)$, grow linearly with each new action. The terms eventually get too big and LPARSE/SMODELS cannot handle them. Some plans, such as changing a tire in TireWorld, are impossible to generate, because the action sequence becomes too long. This problem can be solved by renaming terms, though is a bit messy.

Applications to Theorem Provers

Our research grew out of a desire to write a more controllable theorem prover, particularly for the use of Green style planning. But, even today's theorem provers have cumbersome control mechanisms. The FOL theorem prover OTTER (McCune 1997) allows control through manipulation of weights on literals, which is inadequate. In the meantime, we realized that the ASP paradigm has enough expressivity to write some interesting control rules, such as the concept of *final position*, with minimal coding. From this there came a natural curiosity on how to implement other formalisms, such as *contexts* and *mental situations*, to increase efficiency and at least *conceptually* solve control problems.

In our approach, the stable model semantics is essentially "filling in" facts about situations, through effect axioms and domain constraints. We could in theory use a theorem prover to do the same, if we could restrict it to only deriving facts of the form $Holds(f, result(a, s))$ for a given a and s. Then we could use full blown first order logic to express our action domain, instead of being restricted to logic program rules. (McCarthy 2000) has proposed restricting resolutions on situation variables to a given situation in order to "flesh out" information about a situation.

We should note that ASP and Green style planning are two very different methodologies. ASP is "complete" in the sense that it will compute all models within finite time. Theorem provers on the other hand, can go on forever, and only return a trail of their search. In ASP we achieve direct access to the search for plans since the plan formalization and control rules are on the same level. This solution won't work when applied to a theorem prover. In fact, (Hayes 2000) notes that adding more control information will only add to the search burden of the theorem prover. This phenomenon occurs in our framework as well. Regardless of the methodology, we need some sort of "hook" in order to implement control efficiently, described next.

Future Hopes for Planning Formalisms

We understand that formalizing a control strategy within the answer set planning framework is moot since the underlying computations do not really take advantage of this information. We have achieved some control through the use of *mental situations* and *contexts*, along with controlling the domain of discourse. We hope that this research will will give some insight into the planning problem, so that more sophisticated frameworks can be developed.

For example, eliminating our outer extraction loop, by planning entirely within a chain of mental situations, is impossible due to the strong range restriction of LPARSE. Perhaps a better frameworks can be developed that has built-in support for mental situations, and contexts, including a separate domain of discourse for each mental situation.

For example, we could envision a logic program written as:

```
c1:
 holds(bright(X),S) :-
       holds(is(X,red),S),
       object(X), sit(S).
c2:
 holds(is(X,C),result(paint(C),S)) :-

       holds(prec(paint(C)),S),
       color(C), object(X), sit(S).

mentalsit(m0;m1).

domain(m0,object,[a,b]).
domain(m0,sit,[s0]).
domain(m1,color,[blue,pink]).

in(c1, m0).
in(c1, m1).
in(c2, m1).

inherit(m1,m0).

m0:
 holds(is(X,red),S) :-
       object(X), sit(S).
```

`c1` and `c2` are contexts each containing one rule. `m0` and `m1` are mental situations, and `domain` assigns each a sorted domain of discourse. `in` asserts that only the rules in `c1` apply to the literals that hold in mental situation `m0`, and both `c1` and `c2` apply in `m1`. We could also specify that `m1` `inherit` all the truths of `m0`.

If each mental situation has a different domain of discourse, then we could imagine a computational savings, in that grounding of the rules would be restricted to a subset of the entire domain of discourse. Also, certain rules would not even be used! Having the control mechanism in the same logic as the data is very powerful, in that the mechanism terms are first order objects which can be manipulated themselves. We also note that mental situations could help construct a very intuitive stratification of the logic program, aiding in computation of the answer sets.

Acknowledgments

The author would like to thank Son Cao Tran for his advice and encouragement. Both he and Ale Provetti should be commended for their flexibility as co-chairs. We would also like to thank the referees for the wonderful comments, and critiques. We are grateful to John McCarthy for his support, and identifying these important logical AI concepts, and Vladimir Lifschitz for his introduction to the exciting world of answer set planning.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning[5]. *Artificial Intelligence* 16:123–191.

Bacchus, F. 2000. AIPS 2000 Planning Competition Webpage[6]. Webpage.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs[7]. In *Proceedings of the Fourth European Conference on Planning*, 169–181. Toulouse, France: Springer-Verlag.

Doherty, P., and Kvarnström, J. 1999. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner[8]. In *Proceedings of TIME'99*.

Doherty, P.; Gustafsson, J.; Karlsson, L.; and Kvarnström, J. 1998. Temporal Action Logics (TAL): Language Specification and Tutorial[9]. *Electronic Transactions on Artificial Intelligence*.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*. B.V.: Amsterdam, The Netherlands: Elsevier Science Publishers. 996–1072.

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.

Gelfond, M.; Lifschitz, V.; and Rabinov, A. 1991. What are the limitations of the situation calculus? In Boyer, R., ed., *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Dordrecht: Kluwer Academic. 167–179.

Green, C. 1969. Theorem-proving by resolution as a basis for question-answering systems. In Meltzer, B.; Michie, D.; and Swann, M., eds., *Machine Intelligence 4*. Edinburgh, Scotland: Edinburgh University Press. 183–205.

Hayes, P. 2000. Personal Communication.

Huberman, B. J. 1968. *A Program To Play Chess End Games*. Ph.D. Dissertation, Stanford University. Stanford Artificial Intelligence Project.

Lifschitz, V. 1990. Representing frames in the space of situations. *Artificial Intelligence* 46:365–376.

Lifschitz, V. 2000. Answer set programming and plan generation[10].

McCarthy, J., and Buvač, S. 1998. Formalizing Context (Expanded Notes). In Aliseda, A.; Glabbeek, R. v.; and Westerståhl, D., eds., *Computing Natural Language*, volume 81 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford University. 13–50.

McCarthy, J. 2000. Personal Communication.

McCune, W. 1997. Otter: An Automated Deduction System[11].

Pednault, E. P. D. 1989. Adl: Exploring the middle ground between strips and the situation calculus. In Brachman, R. J.; Levesque, H. J.; and Reiter, R., eds., *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, 324–332. San Mateo, CA: Kaufmann.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* 13 (1–2):81–132.

Sierra, J. 1998. Declarative formalization of strips. In *Thirteenth European Conference on Artificial Intelligence, ECAI-98*.

Subrahmanian, V. S., and Zaniolo, C. 1995. Relating stable models and AI planning domains[12]. In *Proceedings of the Twelfth International Conference on Logic Programming (ICLP'95)*, 233–247. MIT Press.

Syrjänen, T. 2000. *Lparse User's Manual (Draft 1.0)*[13]. Digital Systems Laboratory, Helsinki University of Technology.

[5]http://www.cs.toronto.edu/˜fbacchus/Papers/tlplanaij.ps

[6]http://www.cs.toronto.edu/aips2000/

[7]http://ftp.informatik.uni-freiburg.de/documents/papers/ki/dimopoulos-etal-ecp97.ps.gz

[8]ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time99-final.ps.gz

[9]http://www.ep.liu.se/ea/cis/1998/015/cis98015-revised.ps

[10]http://www.cs.utexas.edu/users/vl/mypapers/asppg.ps

[11]http://www.mcs.anl.gov/home/mccune/ar/otter/

[12]http://www.cs.ucla.edu/˜zaniolo/cz/iclp95.ps

[13]http://www.tcs.hut.fi/Software/smodels/lparse/lparse.ps.gz