Encoding Planning Problems in Nonmonotonic Logic Programs

Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler

Institut für Informatik, Universität Freiburg
Am Flughafen 17, D-79110 Freiburg, Germany
E-mail: <last name>@informatik.uni-freiburg.de

Abstract. We present a framework for encoding planning problems in logic programs with negation as failure, having computational efficiency as our major consideration. In order to accomplish our goal, we bring together ideas from logic programming and the planning systems GRAPHPLAN and SATPLAN. We discuss different representations of planning problems in logic programs, point out issues related to their performance, and show ways to exploit the structure of the domains in these representations. For our experimentation we use an existing implementation of the *stable models* semantics called SMODELS. It turns out that for careful and compact encodings, the performance of the method across a number of different domains, is comparable to that of planners like GRAPHPLAN and SATPLAN.

1 Introduction

Nonmonotonic reasoning was originally motivated by the need to capture in a formal logical system aspects of human commonsense reasoning that enable us to withdraw previous conclusions when new information becomes available. Logic programming systems accommodate nonmonotonic reasoning by means of a form of negation, called *negation as failure* (NAF). In their simplest form, nonmonotonic logic programs (also called normal logic programs) are sets of rules of the form $L \leftarrow A_1, A_2, \ldots, A_n, not\ B_1, not\ B_2, \ldots, not\ B_m$, where $n, m \geq 0$ and L, A_i, B_j are atoms. Atoms prefixed with the *not* operator are called *NAF literals* and can be intuitively understood as follows: *not* B is true iff all possible ways to prove B fail.

However, it is not always clear what "fail to prove" means. Logic programs can exhibit quite complicated structure, especially when some NAF literals depend on other NAF literals. Consider the following program P:

$$a \leftarrow not\ b$$
$$b \leftarrow not\ a$$

Different semantics give different "meaning" to the above program. Two of the most influential semantics for normal logic programs are the *stable models* semantics [6] and the *well-founded* semantics [13]. Under the 2-valued semantics

of stable models, P has two models, one that assigns a the value true and b the value false, and another one that assigns the opposite values. Under the 3-valued well-founded semantics, both a and b are assigned the value *unknown*.
Here we are mainly interested in systems that implement the stable model semantics, but we will also use the well-founded model information for preprocessing and simplifying planning theories. Recent implementations of the stable model semantics include SLG [3], SMODELS [12], and the branch and bound method described in [14].

The relation between nonmonotonic logic programming and reasoning about action has been studied quite extensively in the literature, e.g. by Gelfond and Lifschitz [7]. In fact, there is some work on relating planning and logic programming, for example by using the *Event Calculus* in combination with abduction [4], or, similarly to what we describe here, logic programs and the stable model semantics or its variants [7,5,15]. Most of the research though is concerned more with the representational adequacy of logic programming as a formalism for representing theories of action and less with issues related to computational efficiency.

In this paper we present some preliminary results on representing planning problems in logic programming systems and discuss efficiency issues. For our encodings we borrow from the planning systems GRAPHPLAN [2] and SATPLAN [10]. The basic idea is simple: we encode the planning problems in such a way that the *stable models of the encodings correspond to valid sequences of actions*. Consequently, planning is the problem of finding a stable model that, for a certain time instant t, assigns true to all the fluents that belong to the final state. Action predicates that are true in the stable model and refer to time instants earlier than t, constitute a plan that achieves the goals.

In more detail, we present a number of different encodings of planning problems in logic programming and discuss nonmonotonic reasoning techniques that can be applied to them. Moreover we show how these representations can exploit the structure of the planning domains. Namely, in order to make the representation of the problems more compact we exploit the *post-serializability* property. Roughly speaking, a set of actions is post-serializable if, when applied in parallel and some of their preconditions contradict some of their effects, there is always an order such that if the actions are applied in this order, earlier actions never delete the preconditions of later ones.

We have conducted a number of experiments on problems taken from the planning literature. For these experiments we used SMODELS [12], a recent efficient implementation of the stable model semantics that seems to outperform other existing systems. It turns out that the combination of the above techniques gives an effective planning method. Its performance on a number of hard *blocks-world* and *logistics* problems, compares well with other existing systematic planning methods.

2 Stable Model Semantics and SMODELS

In this section we briefly review the stable model semantics and the SMODELS algorithm. Throughout the paper we assume basic knowledge of logic programming and familiarity with the planning systems GRAPHPLAN [2] and SATPLAN [10]. Due to space limitations we will not discuss the well-founded semantics [13].

A (normal) logic program is a set of rules of the form

$$L \leftarrow A_1, A_2, \ldots, A_n, not\ B_1, not\ B_2, \ldots, not\ B_m$$

where $n, m \geq 0$ and L, A_i, B_j are atoms. We assume that programs are ground, i.e., all atoms are ground. Let M be a set of atoms and P a normal logic program. We define as P^M the Horn program obtained from P by deleting (a) all rules that contain $not\ B_i$ for $B_i \in M$ (b) all NAF literals from the bodies of the remaining rules. The resulting program P^M is a Horn program as no NAF literal occurs in it. The semantics of a Horn program P_h is exactly its minimal model, denoted by $\mathcal{M}(P_h)$.

Definition 1. [6] A set of atoms M is a **stable model** of a normal logic program P iff $M = \mathcal{M}(P^M)$.

Example 2. Consider the following logic program P:
$m \leftarrow a$
$k \leftarrow b$
$a \leftarrow p, not\ b$
$b \leftarrow p, not\ a$
$p \leftarrow$
It is not difficult to see that both $M_1 = \{m, a, p\}$ and $M_2 = \{k, b, p\}$ are stable models of P. The well-founded model assigns true to p and unknown to all other atoms.

The above definition of stable model semantics is not constructive and can only be used to verify whether a set of atoms is a stable model of a program or not. In fact, determining whether a propositional logic program has a stable model is an NP-complete task.

SMODELS is an effective algorithm for computing the stable models of *function-free* normal logic programs. Non-ground programs are first grounded by a parser that is part of the system. The stable models of the resulting propositional program are computed by the SMODELS algorithm[1] that works roughly as follows. At each step, it first chooses a NAF literal to which it assigns a truth value (starting with the value false, meaning that the corresponding atom is excluded from the stable model currently under construction) and then it employs functions that propagate the assumed value in the program and check for conflicts

[1] For all experiments reported here we used SMODELS version 1.5 and the parser version 0.13.

with other values. If an inconsistency is detected, it backtracks and assigns a different value to the literal. Finally, it employs a heuristic for selecting the NAF literal on which it branches. We used this heuristic as it is.

It is important to note that since SMODELS branches only on NAF literals, *the search space consists only of atoms that occur negated in the program.* Atoms that occur only positively in the program are not choice points for the algorithm.

3 Representing Planning Problems

In this section we present the basic domain independent method for encoding plans in nonmonotonic logic programs. To facilitate discussion we assume that we are given a STRIPS-style specification of a planning problem L over a set of fluents F and a set of operators O. Moreover, we assume that the preconditions of the operators contain only positive literals.[2] With L we associate a logic program P_L as follows. For every fluent $fluent_i \in F$, P_L contains a set of rules

$$fluent_i(t) \leftarrow oper_j(t-1) \tag{1}$$

for every $oper_j \in O$ that contains $fluent_i$ in its add effects and every time instant t. For every operator $oper_i$, P_L contains the rule schemata

$$\begin{array}{c} oper_i(t) \leftarrow precon_{i1}(t), \ldots, precon_{im}(t), switchon_i(t), not\ contradict_i(t) \\ contradict_i(t) \leftarrow controp_{i1}(t) \\ \ldots \\ contradict_i(t) \leftarrow controp_{in}(t) \end{array} \tag{2}$$

where $precon_{ij}$ for $1 \leq j \leq m$ are the preconditions of operator $oper_i$, and $controp_{il}$, for $1 \leq l \leq n$, are operators that contradict with $oper_i$. An operator $oper_j$ contradicts with the operator $oper_i$ if the effects of $oper_j$ either contradict the effect of $oper_i$ (i.e., if applied in parallel lead to invalid world states) or are inconsistent with the preconditions of $oper_i$.[3] It is important to note that contradictory operators need not have different predicate names. In the *blocks-world* for example, the operator $move(X, K, Z, T)$ (where the arguments denote, from left to right, the object, the destination, the origin and the time instant) contradicts with $move(X, Y, Z, T)$ for $K \neq Y$, since a block can not move to two different places at the same time. Similarly, $move(L, Y, M, T)$ contradicts with $move(X, Y, Z, T)$ for $L \neq X$ and $Y \neq table$.

The $switchon_i$ predicate implements the choice the system has at each time step t, between applying operator $oper_i$ or keeping it blocked. To realize this effect, we add for each $switchon_i$ predicate the following two rules:

$$\begin{array}{c} switchon_i(t) \leftarrow not\ blocked_i(t) \\ blocked_i(t) \leftarrow not\ switchon_i(t) \end{array}$$

[2] Negative preconditions can be represented as in GRAPHPLAN [2], which adds only polynomial overhead [1].

[3] In GRAPHPLAN terminology this is called operator *interference*. We partly drop this restriction later in the paper by introducing the notion of post-serializable actions.

This pair of rules encodes a local (i.e., not influenced by choices on other literals in the program P_L) decision on the values of $switchon_i$ and $blocked_i$. By the stable model semantics, exactly one of these atoms will be true while the other will be false. If $switchon_i(t)$ gets the value true, operator $oper_i(t)$ can be applied, provided that the rest of the literals in the body of the rule with $oper_i(t)$ in the head are also true.

Finally, we need a set of rules to represent inertia. For each fluent $fluent_i$, P_L contains the rule:

$$fluent_i(t) \leftarrow fluent_i(t-1), not\ changefluent_i(t-1)$$

The rule simply states that a fluent that is true at time $t-1$ remains true at t unless an action changes its value to false. This change is encoded by the NAF literal $not\ changefluent_i(t-1)$ and a set of rules

$$changefluent_i(t) \leftarrow oper_j(t)$$

for every operator $oper_j(t)$ that has $fluent_i$ in its delete effects.

To the above we also add the fluents that are true in the initial state (time t_0), fix a number of time instants $t_0, t_1, \ldots, t_k$ and add type information. In this way we obtain the program P_L. Assume that the final state must contain a set of fluents $F_1, \ldots, F_n$. The planning problem then amounts to finding a stable model M of P_L that assigns true to the atoms $F_1(t_k), \ldots, F_n(t_k)$.

The translation we presented above is not the only possible. In fact, in many cases we can have more compact representations by omitting literals, rules or variables. Consider for instance, part of the encoding we used for the *fixit* domain [2], as depicted in Figure 1. It contains the definition of the rem-wheel ("remove wheel") predicate and some related fluents. In this logic program there is no *switchon* predicate for the rem-wheel operator, but the *blocked* predicate that may block the application of rem-wheel is directly attached to the operator definition. The *prevtime* predicate represents explicitly the relation between time instants. Since SMODELS cannot handle function symbols, all programs contain a set of assertions $prevtime(t_0, t_1)$, $prevtime(t_1, t_2)$, etc.

```
free(Y,T):-hub(Y),wheel(X),prevtime(T,T1),rem-wheel(X,Y,T1).

have(X,T):-hub(Y),wheel(X),prevtime(T,T1),rem-wheel(X,Y,T1).

rem-wheel(X,Y,T):-hub(Y),wheel(X),time(T),high(Y,T),on(X,Y,T),
unfastened(Y,T),not blocked11(X,Y,T).

blocked11(X,Y,T):-hub(Y),wheel(X),time(T),not rem-wheel(X,Y,T).

free(Y,T):-hub(Y),wheel(X),prevtime(T,T1),free(Y,T1),not occ(Y,T1).

occ(Y,T):-hub(Y),wheel(X),time(T),put-wheel(X,Y,T).
```

Fig. 1. The *fixit* domain

Moreover, by modifying slightly the above encoding it is possible to reduce the number of NAF literal. For instance, instead of representing operator interference through the rule schema (2) we can write:

$$\begin{array}{c} oper_i(t) \leftarrow precon_{i1}(t), \ldots, precon_{im}(t), switchon_i(t) \\ inco \leftarrow oper_i(t), controp_{i1}(t) \\ \ldots \\ inco \leftarrow oper_i(t), controp_{in}(t) \end{array} \qquad (2')$$

and compute a stable model where the facts of the final state are true, but *inco* is false. In this way we prohibit the parallel execution of contradicting actions, and avoid including in the program the NAF literal *not* $contradict_i(t)$ of rule schema (2). We call these encodings *constraint-based*. There are also other ways to "optimize" the logic programming representation of planning problems, which we will not discuss here due to space limitations.

The method we put forward differs from the encoding of planning problems in satisfiability proposed by Kautz and Selman [9], [10]. It is different from the propositional theories of [9] in that the logic program representation explicitly encodes contradictions between operators through the rule schemata (2). Moreover, it does not include axioms stating that actions imply their preconditions. The set of rules (2) and the *minimality* of the stable models suffice to ensure that an operator is applicable only if its preconditions hold. The logic programming representation is also different from the GRAPHPLAN-based encoding [10] in that it uses frame axioms instead of no-ops to solve the frame problem.

More importantly, the search spaces of SMODELS and propositional logic encodings are different. On one hand, the logic programming encoding introduces new predicate symbols in the problem representation (e.g., the *block* or *switchon* predicates) that do not appear in a STRIPS-style problem description. But on the other hand, recall that the search space for the SMODELS algorithm consists of NAF literals only.

4 Computing with Logic Programs

In this section we describe a number of different encodings of planning problems, discuss some efficient pre- and post-processing methods, and report on a number of experiments we conducted with the SMODELS system.

4.1 Linear Encodings

Recall that SMODELS computes the stable models of ground logic programs, therefore its performance (similar to GRAPHPLAN and SATPLAN) depends crucially on the number and the arity of the predicates of the input theory. For domains that contain predicates with high arity and variables with large domains, grounding can result in prohibitively large theories (here ground logic programs). Kautz and Selman [9] describe a method, called linear encoding, that splits operators with many arguments into a number of predicates with less

```
on(X,Y,T1):-prevtime(T1,T2),diff(X,Y),mvable(X),on(X,Y,T2),not move-obj(X,T2).
on(X,Y,T1):-prevtime(T1,T2),mvable(X),diff(X,Y),diff(X,Z),diff(Z,Y),
                on(X,Z,T2),move-obj(X,T2),move-dest(Y,T2).
clear(X,T):-mvable(X),time(T),not occ(X,T).
occ(X,T):-mvable(X),mvable(Y),time(T),on(Y,X,T).
move-obj(X,T):-mvable(X),clear(X,T),not otherobj(X,T),not blocked(X,T).
otherobj(X,T):- mvable(X),mvable(Y),diff(X,Y),move-obj(Y,T).
blocked(X,T):-mvable(X),not move-obj(X,T).
move-dest(X,T):-clear(X,T),mvable(Y),move-obj(Y,T),diff(X,Y),not otherdest(X,T).
otherdest(X,T):-diff(X,Y),move-dest(Y,T).
```

Fig. 2. *Blocks-world* with linear encodings

(in fact two) arguments. With this encoding, only one action can be applied at each time. Obviously, it is possible to obtain similar encodings for logic programming representations. Such a program for the *blocks-world* domain is depicted in Figure 2. Note that the representation explicitly requires that only one move action can be applied at each time.[4]

If the number of time steps is set to the length of the optimal plan, SMODELS is able to solve within reasonable time some hard *blocks-world* instances which are variants of those introduced in [10] (see Table 1). However, the algorithm seems to be quite sensitive to the "details" of the encoding. For instance, by replacing the rules for *move-obj*, *otherobj* and *blocked* of Figure 2 with the rules:

$move\text{-}obj(X,T) : -mvable(X), time(T), clear(X,T), not\ blocked(X,T).$
$blocked(X,T) : -mvable(X), mvable(Y), diff(X,Y), time(T), move\text{-}obj(Y,T).$
$blocked(X,T) : -time(T), mvable(X), not\ move\text{-}obj(X,T).$

the runtime for the problem `bw-large.c` [10] increases by 40%. Nevertheless, the largest difference observed in the runtimes was never more than 70%. The phenomenon seems to be related to the heuristic used by the algorithm to select the branch literals.

4.2 Using the Well-Founded Semantics to Prune the Representation

The well-founded semantics [13] is essentially a 3-valued model that always exists, is unique for every logic program, and is traditionally considered to be a polynomial time "approximation" of the stable model semantics. Whenever the well-founded semantics assigns the value true to an atom, this atom will be true in all stable models and symmetrically, all atoms that are false in the well-founded model will be false in all stable models. The atoms that are not assigned

[4] A move action at time t is represented by *move-obj* and *move-dest* at time t.

a value in the well-founded model (unknown atoms) can have different values in different stable models. Although the well-founded semantics is too weak for planning problems (it is eager to assign the value unknown whenever confronted with a choice) it can be a useful and cheap preprocessing step that can reduce the size of planning problems.

To see how it works, consider a *blocks-world* problem with initial state $on(A, table)$, $on(B, A)$, $on(C, B)$. Clearly blocks B and A are occupied at time t_0 and the well-founded semantics will assign true to $occ(A, t0)$ and $occ(B, t0)$. Then, by the frame axiom (regardless of where C moves) $on(B, A), on(A, table)$ hold for time $t1$. Consequently $occ(A, t1)$ and $on(A, table, t2)$ hold. Therefore, we can omit ground rules and atoms that are not consistent with the above information, for instance we can omit the ground literal $on(A, C, t1)$ and all ground rules in which it occurs.

Using the information provided by the well-founded model, we can produce smaller ground instances. For the `bw-large.c` problem [10] for instance, the number of atoms reduces from 5,101 to 4,558 and the number of rules from 58,201 to 47,729 (see also Table 1). Computing the well-founded model never takes more than a few seconds.

4.3 Parallel Steps, Post-Serializability, and Weak Post-Serializability

Although linear encodings give compact representations, they have the disadvantage that they do not allow for parallel actions. Therefore, the number of time steps of a plan equals the number of actions in this plan. If we abandon linear encodings and adopt parallel actions, the arity of the operator predicates increases but at the same time we may obtain plans that achieve the goals in considerably fewer time steps. Therefore, it may happen that the size of the ground logic program also decreases and, more importantly, finding a solution becomes easier.

Clearly, the encoding we presented in Section 3 allows for parallel steps. For some domains however, it is possible to gain, by exploiting their structure, more parallelism during plan generation. First, we define some necessary notions. For a set of actions A, we define the *preconditions-effects graph* of A, denoted by A_G, to be the graph that contains a node for each action in A, and an edge from an action a_i to an action a_j if the preconditions of a_i are inconsistent with the effects of a_j.

Definition 3. A set of actions A that refer to the same time instant is *post-serializable* if (a) the union of their preconditions is consistent (b) the union of their effects is consistent and (c) the graph A_G is acyclic.

If a domain contains a set of post-serializable actions A, then we can apply these actions in parallel at a time instant t during plan generation, and then serialize them during a post-processing phase that works as follows: index with time t all actions that have in-degree 0 in A_G, remove these actions form A_G and index all actions that have in-degree 0 in the new graph with time $t + 1$. Repeat

this procedure until all actions are removed and assume that the last time index used is $t+k$. Then assign time $t+k+1$ to all actions that have been applied at time $t+1$ in the original plan. In the next section, we will see some domains where the post-serializability property holds.

Consider now the *blocks-world* domain and suppose that we want to find plans that achieve goals expressed in terms of the *on* predicate. Assume that we encode the domain in a logic program that disables the operators $move(X, M, Z)$ and $move(K, Y, L)$ whenever the operator $move(X, Y, Z)$ is applicable, but it does not disable the operator $move(N, X, P)$. Clearly, this domain representation enables parallel execution of actions that are not post-serializable. Consider for example two blocks A and B that are on the table and clear. Then, the actions $move(A, B, table)$ and $move(B, A, table)$ can be applied in parallel, but they create a cycle since each of them deletes the *clear* precondition of the other. However, for planning problems that involve such sets of parallel actions, the *weak post-serializability* property may hold.

A set of actions A is *weakly post-serializable* if for the actions of A that refer to the same time instant the following conditions hold: (a) the union of their preconditions is consistent (b) there exists a function that maps the nodes of every possible cycle of A_G into a consistent set of acyclic actions such that all other actions of A remain valid with respect to the new set of actions and (c) if A achieves a consistent goal G, the new set of actions also achieves G.

Intuitively, a set of actions A is weakly post-serializable if whenever a cycle occurs in A_G, there is a way to break the cycle by replacing some of the actions in the cycle by other actions. Therefore, if the actions of a plan are weakly post-serializable, we can transform these actions into a valid plan, if one exists. Note that the action replacement must be local, i.e., no other actions of the plan must be affected. The way that cycles can be removed in the *blocks-world* domain is simple: move to the table one or more of the blocks involved in the cycle. This is a local replacement. Blocks that form a cycle cannot be cleared by any future actions and therefore they do not affect the rest of the plan. Figure 3 displays a weakly post-serializable move operator for the *blocks-world* domain. Observe that there is no rule that prohibits a block to move and at the same time another block to move on top of it[5]. Therefore in a state where blocks A, B and C are clear, the operators $move(A, B, t_i)$ and $move(B, C, t_i)$ can be applied in parallel. In the post-processing phase, the second action will be ordered before the first.

In terms of the logic programming approach we use here, all the above mean that it may be possible to remove from the representation some of the contradictions between the operators, and push the conflict resolution into the post-processing phase. This enables the planner to shorten plan length, and find an initial solution in fewer steps. This solution is then transformed, by post-processing, into the final plan.

[5] Observe also that the source of a block that moves is not specified in the move operator. This information can be easily derived in the post-processing phase from the *move* and *on* predicates.

```
move(X,Y,T):-moveable(X),time(T),block(Y),diff(X,Y),diff(X,Z),diff(Z,Y),
on(X,Z,T),clear(X,T),clear(Y,T),switchon(X,T),not other(X,Y,T).

other(X,Y,T):-moveable(X),time(T),block(Y),diff(X,Y),diff(Y,Z),move(X,Z,T).

other(X,Y,T):-moveable(X),moveable(Y),time(T),block(Y),diff(X,Y),diff(X,Z),
diff(Y,Z),move(Z,Y,T).

switchon(X,T):-moveable(X),time(T),not blocked(X,T).

blocked(X,T):-moveable(X),time(T),not switchon(X,T).
```

Fig. 3. *Blocks-world* with weakly post-serializable move operator

It turns out that action parallelism can have serious mitigating effects on the computation. For the 15 blocks problem `bw-large.c`, for instance, the method achieves the goals in 6 time steps (contrast this with the 14 steps of the linear encoding). Although in the parallel encoding the number of atoms and rules increases slightly, the overall runtime is smaller. The combination of parallel representation with well-founded model preprocessing reduces further the computation time (see Table 1). For the same problem, GRAPHPLAN with operators that allow parallel move actions, needs 31 minutes and 8 time steps. If we increase the number of blocks by two (problem `bw-large.d`), GRAPHPLAN finds a solution after 61 hours. The even larger problem `bw-large.e` with 19 blocks is practically unsolvable for GRAPHPLAN. The runtimes for SMODELS are given in Table 1. The rows showing GP-parallel in this table refer to an encoding that allows as much parallelism as GRAPHPLAN does. Note that it can be the case that grounding is more expensive than finding a solution.

Problem	Time/Actions	Atoms	Rules	Time
`bw-large.c` (linear)	14/14	5,101	58,201	1,482 (125)
`bw-large.c` (linear/well-found.)	14/14	4,558	47,729	1,110
`bw-large.c` (parallel)	6/20	5,572	65,851	483 (200)
`bw-large.c` (parallel/well-found.)	6/21	4,404	46,126	190
`bw-large.d` (parallel/constraint)	6/32	8,138	85,101	157 (639)
`bw-large.d` (GP-parallel/constraint)	9/36	11,874	169,102	450 (1,976)
`bw-large.e` (parallel/constraint)	7/37	11,623	139,499	365 (1,568)
`bw-large.e` (GP-parallel/constraint)	10/44	16,211	260,700	1,216 (4,270)
`logistics.c` (parallel)	8/68	2,529	10,531	18 (23)
`trains.a` (parallel/constraint)	8/39	1,957	7,786	647 (14)
`trains.b` (parallel/constraint)	7/34	2,234	13,746	1,261 (17)
`trains.c` (parallel/constraint)	8/42	2,514	15,942	5,989 (22)

Table 1. Runtimes for solving planning instances on a SUN ULTRA with 256M RAM. Times for grounding/parsing are given in brackets. All times are in CPU seconds. SMODELS was run with the *lookahead* option on.

4.4 Other Domains

We have tested the method on a number of other domains in order to obtain a more complete idea of its performance. The *fixit* domain, as it appears in GRAPHPLAN's distribution, is interesting because it contains a fairly large number (in fact 13) of operators and part of the actions are post-serializable (e.g., putting something in the boot and closing the boot at the same time). The performance of SMODELS (without well-founded preprocessing solved in 0.23 sec) is comparable to GRAPHPLAN (0.11 sec).

A more interesting domain is the *logistics* domain [10]. A number of packages are in different places in the initial state and the task is to deliver these packages to destinations specified in the final state, using the available resources (trucks and planes). This domain also contains post-serializable actions. The actions of loading a package to a truck/plane, unloading the package from a truck/plane and driving the truck/flying the plane can occur at the same time. During post-processing all drive/fly actions will be delayed for one time step after the load and unload action with which they conflict.

This domain is hard even for GRAPHPLAN. GRAPHPLAN was not able to solve the problem `logistics.c` after running for 6 CPU hours[6]. If the number of time steps is set to the length of the optimal plan, SMODELS with parallel encoding and without well-founded model preprocessing solves this problem in 18 seconds. Part of the efficiency of the method can be attributed to the use of simple constraints that, although implied by the rules, help the algorithm to detect dead-ends earlier. These constraints are:
$inco : -veh(Y), obj(X), veh(Z), diff(Y,Z), in(X,Y,T), in(X,Z,T).$
$inco : -obj(X), loc(Y), loc(Z), diff(Y,Z), at(X,Y,T), at(X,Z,T).$
$inco : -veh(X), corloc(X,Y), corloc(X,Z), diff(Y,Z), at(X,Y,T), at(X,Z,T).$
$inco : -obj(X), loc(Y), at(X,Y,T), veh(Z), in(X,Z,T).$
where the predicate *corloc* contains the possible location of the vehicles. Like in a constraint-based encoding, SMODELS is asked to compute a stable model where the facts of the final state are true and *inco* is false. If during search, the assumed values derive *inco*, the algorithm backtracks immediately. Finally note that due to the high branching factor of the domain the well-founded model information is not of much help.

Similar to the *logistics* is the *trains* domain [8]. We are given a number of cars that can carry commodities and a number of engines that can be coupled with cars and move between cities that are connected by tracks. The task is to carry the commodities to their destinations. We can also require that the cars/engines are in specified locations in the final state. The main feature of the domain, in its UCPOP description, is a conditional *move* operator: if a car is coupled to an engine, it moves with the engine. Again, our encoding contains constraints similar to those of the *logistics* domain. We give some representative runtimes

[6] Kautz and Selman [10] report that for Walksat and *state-based encodings* the problem is solved in 1.9 seconds on a SGI Challenge/150 MHz. For other encodings and algorithms the problem is unsolvable (needs more than 10 hrs of CPU time).

for this domain in Table 1. The interesting point here is that although these theories are relatively small, they can be very hard for SMODELS.

The last set of experiments concerns the towers of Hanoi domain. SMODELS can solve the 4 blocks problem in a few seconds but the 5 blocks problem seems to be beyond its capabilities, at least for our encoding. Even though we have not applied the well-founded model preprocessing that can presumably reduce the complexity, the domain seems to have features that make it hard for the method; action parallelism is low and the constraints to be satisfied are very tight.

4.5 Other Issues

Finding a correct plan is not the only issue. Minimizing the number of actions is also important. This becomes especially important for systems like SMODELS that view planning as a constraint satisfaction problem. In fact, the plans synthesized by SMODELS often contain obviously redundant actions. This raises the question of what are good methods that remove redundant actions from plans. It seems that for domains like the *blocks-world* the problem is harder than for domains similar to *logistics*.

One possible way around this problem could be to give SMODELS' output, i.e., the set of actions that achieves the goal, as input to an efficient planner like GRAPHPLAN. If the number of (ground) operators (i.e., the actions in the initial plan) is small, it is possible that the planner will find a solution quickly and at the same time remove some of the redundant actions. The idea is related to the methods for ignoring irrelevant facts and operators during plan generation developed by Nebel *et al.* [11].

When using logic programming representations, we are confronted with another important issue. The run times we report above are for logic program representations with the number of time steps set to the length of the optimal plan. For practical problem solving, we can use binary search over the length of the plan [10] (or a similar method, depending on the size of the problem). This raises the question of how fast the algorithm fails in cases where the allowed plan length is less than necessary. For the *blocks-world* domain and the parallel encoding, the well-founded model information helps the system to determine unsolvability in a few seconds. Similarly for the *trains* problems of Table 1. For the *logistics* domain, proving unsolvability is harder, but still the difference between the performance of the method and GRAPHPLAN or SATPLAN is dramatic. For the `logistics.c` problem, if the allowed length is set to one less than the minimum, SMODELS reports that no stable model exists in 1,447 secs.

5 Conclusions

We presented techniques for encoding planning problems in nonmonotonic logic programs. We have provided some initial evidence that the combination of ideas from nonmonotonic reasoning and planning may deliver effective planning systems. We have also shown that planning problems constitute an interesting and

challenging set of benchmarks for nonmonotonic reasoning system implementations.

In the future, we intend to work on a tighter integration of planning and nonmonotonic reasoning methods. One issue is how techniques used in GRAPHPLAN, like the automatic derivation of exclusive pairs, can be captured in the logic programming framework. Moreover, it is still open whether the branching heuristic can be modified in a way that it becomes more effective for logic programs that correspond to planning problems.

Acknowledgments

We would like to thank Alfonso Gerevini, Bertram Ludäscher and Ilkka Niemelä for their feedback.

References

1. C. Bäckström. Equivalence and tractability results for SAS^+ planning. *KR-92*.
2. A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, Vol. 90(1-2), 1997.
3. W. Chen and D.S. Warren. Computation of Stable Models and its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, to appear.
4. M. Denecker, L. Missiaen and M. Bruynooghe. Temporal Reasoning with Abductive Event Calculus. *ECAI-92*.
5. P. M. Dung. Representing Actions in Logic Programming and its Applications in Database Updates. *ICLP-93*.
6. M. Gelfond and V. Lifschitz. The Stable Models Semantics for Logic Programs. *ICLP-88*.
7. M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, Vol. 17, 1993.
8. A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research*, Vol. 5, 1996.
9. H. Kautz and B. Selman. Planning as Satisfiability. *ECAI-92*.
10. H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. *AAAI-96*.
11. B. Nebel, Y. Dimopoulos and J. Koehler. Ignoring Irrelevant Facts and Operators in Plan Generation. *ECP-97*, this volume.
12. I. Niemela and P. Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. *International Joint Conference and Symposium on Logic Programming*, 1996.
13. A. Van Gelder, K. Ross and J. Schlipf. The Well-founded Semantics for General Logic Programs. *Journal of the ACM*, Vol. 38, 1991.
14. V.S. Subrahmanian, D. Nau and C. Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, 1995.
15. V. S. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. *ICLP-95*.