The Generation of Pre-Interpretations for Detecting Unsolvable Planning Problems

D.A. de Waal
Department of Computer Science and Information Systems,
Potchefstroom University for Christian Higher Education,
Private Bag X6001, Potchefstroom, 2531, South Africa
Tel: +27 (0148) 299 2535, Fax: +27 (0148) 299 2799
RKWDADW@puknet.puk.ac.za

M. Denecker M. Bruynooghe
Departement Computerwetenschappen, Celestijnenlaan 200A,
Katholieke Universiteit Leuven, B-3001 Heverlee, Belgium
{marcd,maurice}@cs.kuleuven.ac.be

M. Thielscher
Fachgebiet Intellektik, Fachbereich Informatik,
Technische Hochschule Darmstadt, Alexanderstrasse 10,
D-64283 Darmstadt, Germany
mit@intellektik.informatik.th-darmstadt.de

December 10, 1998

Keywords: Deductive Planning, Model Generation, Abstract Interpretation

1 Introduction

In deductive planning, there exists a particular class of planning problems, called unsolvable planning problems, which although successfully treated theoretically, turns out to be undetectable using ordinary resolution methods as infinite sequences of useless actions have to be considered. These problems are exacerbated when we extend our problem domains to infinite domains of resources, e.g. a blocks world with a ("theoretically") unlimited number of blocks or an electric circuit with an unlimited number of switches. Furthermore, the introduction of new objects, by means of "generating" actions (see Section 4), poses huge problems to most if not all existing planning approaches. This "discrepancy" between theory and practise is annoying and prevents us from deciding interesting unsolvable planning problems. Our aim in this paper is therefore to develop an analysis method specifically for the detection of such problems.

Winston in [10] gives an algorithm that, amongst other things, also detects some impossible plans. His algorithm is a search procedure that extends *partial plans* into *complete plans*. Although the algorithm may detect some impossible plans (unsolvable planning problems) it relies on a loop detection mechanism to detect failure. Furthermore, it tries to extend partial plans (starting from the empty plan) to complete plans by enumerating all partial plans. If we now have a planning problem with an infinite number of partial plans of which none can be extended to a complete plan, we clearly have a problem as the algorithm will not terminate.

In [2] we also investigated the detection of unsolvable planning problems using logic programming analysis and transformation techniques [1]. In our approach, a logic program was written [2] implementing the equational logic programming approach to deductive planning [5, 6]. This logic program was first specialized with respect to a particular goal (defining an unsolvable planning problem) with partial evaluation [8]. Second, the result of partial evaluation was approximated using a regular approximation tool [4]. As failure is decidable in a regular approximation, we can test if the goal we are interested in fails in the approximation. If this is the case, we have detected an unsolvable planning problem.

The presented approach however suffered from the imprecision caused by a loss of argument dependency information in the analysis, e.g. in the Blocks World [10] the set of terms $\{on(a,b),\ on(b,a)\}$ is approximated by the superset $\{on(a,a), on(a,b), on(b,a), on(b,b)\}$. This causes a considerable loss of information and prevents the detection of some interesting unsolvable planning problems. Other domains in planning, such as the Kitchen domain used in [9], also have binary functors and a similar loss of argument dependency information may be experienced when analyzing these domains with similar approximation systems. Furthermore, regular approximations are notoriously bad at counting, as they usually count $0, 1, many$ or even $0, many$. This property of regular approximations makes it difficult to keep track of the number of blocks in a blocks world or the number of resources occurring in a problem.

In our approach, the intended semantics of a planning problem is assumed to be given by its least Herbrand model. The idea is to compute a finite abstract model that is a safe approximation of the least Herbrand model of the deductive planning problem. For a large class of formulas, falsity of a formula in the abstract model implies falsity in the least Herbrand model. We show how this approach can be used to decide that interesting planning problems are unsolvable.

The construction of the abstract model and checking of failure in the abstract model is done using available techniques [3]. The method is based on augmenting a logic program, representing a planning problem, with "denotes" predicates that implements a generated pre-interpretation. An immediate consequence operator (similar to T_P) is then used to construct the Least Abstract Model (also called the Least non-Herbrand Model in [3]) of the program. Interesting program properties can be proved by interpreting the results generated in the Least Abstract Model. In this context, we reformulate their results solely for proving failure. No interpretation of the results generated by their method is needed beyond the checking of failure in the abstract model and transferring the result to the least Herbrand model (see Section 3 for more details).

The rest of this paper is organized as follows. First, we explain what comprises an unsolvable planning problem. The model-based analysis is explained in Section 3. In Section 4 we demonstrate the detection of unsolvable problems with two examples. The automation of the proposed method is investigated in Section 5. We conclude with a brief discussion.

We assume the reader is familiar with the notions of a first-order language, pre-interpretation, interpretation and model as defined in [7].

2 Unsolvable Planning Problems

Before we can define an unsolvable planning problem, we recall the logic program from [2] implementing the equational logic programming approach to deductive planning [5, 6]. The original approach employs a specific equational theory, called AC1 [6], to formalize situation descriptions, which, essentially, are multisets of resources that are available in the situation. In contrast, the following program represents situations by lists of resources, and the matching operation with respect to AC1 is encoded via additional clauses. Hence, the program is executable using SLD-resolution.

```
causes(I,void,I).
causes(I,plan(A,P),G) :- action(C,A,E),
                         ac1_match(C,I,Z),
                         append(E,Z,S),
                         causes(S,P,G).

ac1_match(S,T,Z) :- mult_subset(S,T,Z).

mult_subset([],T,T).
mult_subset([E|S],T,R) :- mult_minus(T,E,T2),
                          mult_subset(S,T2,R).

mult_minus([E|R],E,R).
mult_minus([E|R],E1,[E|R1]) :- mult_minus(R,E1,R1).

append([],X,X).
append([X|Xs],Y,[X|Zs]) :- append(Xs,Y,Zs).
```

An instance of $causes(I, J, K)$ is true if the plan J (sequence of action names) transforms an initial situation I into a final situation K[1]. The predicate $action(C, A, E)$ defines the action descriptions of our deductive planning problems where C and E are respectively the *condition* and *effect* (multisets of resources represented by lists) and A is the name of the action. Such an action is applicable in a situation if the latter contains condition C, and the resulting situation is obtained by removing the resources in C from the situation and adding the resources in E.

The resulting program is a definite logic program. The intended semantics of the planning program P is given by its Least Herbrand Model M_{LH}. A planning problem can then be formulated as

```
?- causes(I,P,G),ac1_match([l1,...,ln],G,Z).
```

where I is a multiset of resources representing the initial situation and G the final situation containing resources $[l1, \ldots, ln]$ (resources may contain variables, but variables as resources are not allowed).

A solution to a planning problem is given by a substitution θ such that

$$M_{LH} \models \forall(causes(I, P, G), ac1\_match([l1, ..., ln], G, Z)\theta).$$

A planning problem is unsolvable iff no answer substitution exists, i.e.

$$M_{LH} \models \forall\neg(causes(I, P, G), ac1\_match([l1, ..., ln], G, Z)).$$

No finite SLD-tree will therefore exist when we have an unsolvable planning problem.

It would have been simpler if we could only allow queries of the form

```
?- causes(I,P,G).
```

However, using only *causes* as goal restricts the queries we can state as it is impossible to write that resources $l1, \ldots, ln$ are included in the effect. Furthermore, it is clear that to check if our planning problem is unsolvable we only need to examine the result computed for $causes(I, P, G)$ in the abstract model, although the given query provides the general form of queries allowed in our planning problems. For simplicity of presentation we restrict ourselves to the case $n = 1$.

3 Model-Based Analysis

The intended semantics of a planning program P is given by its Least Herbrand Model M_{LH}. This means that for a given query formulated as a first order

---

[1] It is assumed that we have complete information about relevant facts in the initial situation.

sentence F, we are interested whether or not F is true or false in M_{LH}, i.e. whether $M_{LH} \models F$ or $M_{LH} \models \neg F$.

Our technique is based on the generation of a model in which a sentence F is false. The existence of such a model entails that F is not logically implied by P, i.e. $P \not\models F$. In general, this sort of answer is weaker than the intended answer, namely that $M_{LH} \models \neg F$. However, due to the minimality of the Least Herbrand Model, it holds for a large class of sentences F that $M_{LH} \models F$ iff $P \models F$. The following theorem asserts this.

Theorem 3.1 *Given is a definite program P with Least Herbrand Model M_{LH} and a sentence F containing only the connectives $\wedge, \vee$ and $\exists$, then $P \models F \Leftrightarrow M_{LH} \models F$.*

Hence, by constructing a model M of P in which $M \not\models F$, the theorem allows to conclude safely that $M_{LH} \not\models F$, or equivalently that $M_{LH} \models \neg F$. The model we construct is the least model according to some well chosen pre-interpretation. The queries used to formulate a planning problem satisfy the syntactic restrictions of Theorem 1.

4 Generating Pre-Interpretations

A pre-interpretation consists of a domain and, for each function, a mapping over domain elements. We first illustrate the selection of a pre-interpretation with an example unsolvable problem. The problem is from the classical Blocks World domain [10], but the description is augmented with two action description that can add two blocks or delete two blocks from our blocks world (see action descriptions (5) and (6) below). The robot arm may hold (or not hold) a block, represented by $ho(V)$ and em respectively, where the variable V represents a block. A block may be on a table or on top of another block, represented by $ta(V)$ and $on(V, W)$ respectively, where V and W represent blocks. A block V is clear if there are no other blocks on it. This is represented by $cl(V)$.

```
action([ho(V)],put_down(V),[ta(V),cl(V),em]).                          (1)
action([cl(V),ta(V),em],pick_up(V),[ho(V)]).                           (2)
action([ho(V),cl(W)],stack(V,W),[on(V,W),cl(V),em]).                   (3)
action([cl(V),on(V,W),em],unstack(V),[ho(V),cl(W)]).                   (4)
action([on(V,W),cl(V),em],add_two,
       [on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V)),em]).           (5)
action([on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V)),em],delete_two,
       [on(V,W),cl(V),em]).                                            (6)
```

We now have a more complex problem description than is normally the case in Blocks World problems, as the number of blocks is not fixed: we have therefore named ("numbered") the blocks using the successor function, e.g.

$0,\ s(0),\ s(s(0)),\ \ldots$. This greatly increases the complexity of unsolvable problems in this domain (problems may become undecidable). We can also think of this problem description as describing a blocks world with a block dispensing machine or a block manufacturing machine that can produce and recycle blocks (two at a time) and a robot hand that can stack and unstack the blocks on a table.

Note that it is impossible to represent an infinite number of resources directly in our current framework without resorting to ad hoc procedures, as we are only able to represent finite multisets of resources using the standard list notation. These action descriptions therefore illustrate a general method when attempting to reason over infinite domains of resources. Define one or more action descriptions to implement a generator procedure that can systematically generate successive resources in some domain. Although at any stage of the planning process, only a finite number of resources can be generated, this method is powerful enough to model most problems: an infinite number of situations with a finite number of resources in each situation can be generated. Our second example further illustrates this point.

A query that we may be interested in is[2]:

```
?- causes([on(s(0),0),ta(0),cl(s(0)),em],Plan,
          [on(s(s(0)),s(0)),on(s(0),0),ta(0),cl(s(s(0))),em]).
```

This is obviously an unsolvable planning problem as we start off with an even number of blocks in our initial situation and our final situation requires an odd number of blocks[3]. Detecting this however is not straightforward as we have a possibly increasing (or decreasing) number of resources that are being produced (or consumed).

What we are aiming at is a pre-interpretation where the model of *causes* is

$$causes(good,\ \_,good) \qquad\qquad causes(bad,\ \_,bad)$$

and where our query is mapped to $causes(bad,\ \_,good)$ or $causes(good,\ \_,bad)$.

Note that the plan is ignored as it does not contribute to detecting that our problem is unsolvable: it can therefore be mapped to *good* or *bad* without influencing the rest of the discussion. Because $causes(bad,\_,good)$ (or $causes(good,\_,bad)$) is false in the generated abstract model, our query is not a logical consequence of our program P and therefore false in the Least Herbrand Model of P. We have detected an unsolvable problem.

In the rest of this section we explore ways of generating a pre-interpretation such that we get a model similar to the one sketched above. Our query

```
?- causes([on(s(0),0),ta(0),cl(s(0)),em],Plan,
          [on(s(s(0)),s(0)),on(s(0),0),ta(0),cl(s(s(0))),em]).
```

---

[2] This query can easily be rewritten in the form given in Section 2.

[3] This example is kept simple to aid understanding.

will be used as a starting point of our analysis.

An examination of our query shows that $[on(s(0),0), ta(0), cl(s(0)), em]$ must be mapped to *good* and $[on(s(s(0)),s(0)), on(s(0),0), ta(0), cl(s(s(0))), em]$ to *bad* (or vice versa). Because we are interested in constructing the simplest pre-interpretation sufficient for detecting that our query (actually, the mapping of our query using the generated pre-interpretation) is false in the generated abstract model, we try to generate as few domain elements in the domain of pre-interpretation as possible. To achieve this aim, we examine the "difference" between the two lists of resources occurring in our query. If we for the moment only concentrate on the outer functors occurring in resources, our starting situation has one *on* and our final situation two (*ta*, *cl* and *em* occur in equal numbers in the starting and final situations). *on* is therefore mapped to *good* and *ta*, *cl* and *em* to *bad*. What remain is to fill in the mappings for concatenation on lists:

$$\begin{array}{ll} [good|good] \rightarrow ? & [good|bad] \rightarrow ? \\ [bad|good] \rightarrow ? & [bad|bad] \rightarrow ? \end{array}$$

An examination of the final situation in our query forces the following mapping:

$$[good|good] \rightarrow bad$$

(the mappings of the starting and final situations must be different). Furthermore, we do not want the resources that occurred in equal numbers in the starting and final situations (not part of the difference in starting and final situations) to change our mappings. This forces the following mappings:

$$\begin{array}{ll} [X|bad] \rightarrow X & [bad|X] \rightarrow X \end{array}$$

where X is a variable. As the only domain elements we have so far are *good* and *bad*, we instantiate X to *good* and *bad* respectively to get the following three mappings:

$$\begin{array}{lll} [good|bad] \rightarrow good & & [bad|good] \rightarrow good \\ [bad|bad] \rightarrow bad & \text{(2x)} & \end{array}$$

The only outstanding mappings are that of the empty list $[\,]$ and *ho* (it does not occur in our query but in an action description). The same reasoning applies here, in that we do not want the empty list, denoting no resources, and *ho* to change our mappings ($[\,]$ and *ho* therefore falls into the same catagory as *ta*, *cl* and *em*). The following mappings therefore results:

$$\begin{array}{ll} [\,] \rightarrow bad & ho \rightarrow bad \end{array}$$

If we now map all the numbered blocks to a domain element *block*, we have constructed the following pre-interpretation over $D = \{block, good, bad\}$:

$0 \to \mathit{block}$
$[\,] \to \mathit{bad}$
$\mathit{ho}(\mathit{block}) \to \mathit{bad}$
$\mathit{on}(\mathit{block}, \mathit{block}) \to \mathit{good}$
$[\mathit{good}|\mathit{good}] \to \mathit{bad}$
$[\mathit{good}|\mathit{bad}] \to \mathit{good}$
$s(\mathit{block}) \to \mathit{block}$
$\mathit{em} \to \mathit{bad}$
$\mathit{ta}(\mathit{block}) \to \mathit{bad}$
$\mathit{cl}(\mathit{block}) \to \mathit{bad}$
$[\mathit{bad}|\mathit{good}] \to \mathit{good}$
$[\mathit{bad}|\mathit{bad}] \to \mathit{bad}$

Without any further complications we can compute the finite abstract model based on this pre-interpretation. Unfortunately, the model of *causes* is

$\mathit{causes}(\mathit{good}, \_, \mathit{good})$
$\mathit{causes}(\mathit{bad}, \_, \mathit{good})$
$\mathit{causes}(\mathit{good}, \_, \mathit{bad})$
$\mathit{causes}(\mathit{bad}, \_, \mathit{bad})$

and we have failed to detect that the problem is unsolvable as $\mathit{causes}(\mathit{good}, \_, \mathit{bad})$ is true in this model.

As a second step, we have to "debug" the constructed pre-interpretation to try and determine why

$\mathit{causes}(\mathit{good}, \_, \mathit{bad})$
$\mathit{causes}(\mathit{bad}, \_, \mathit{good})$

are also in our abstract model.

¿From action description (3) we can see that the condition is mapped to *bad* and the effect to *good*. The execution of this action description has as result the generation of one of the unwanted *causes* consequences in our abstract model. As we want action descriptions to preserve mappings (an action description with *good* in the condition must also have *good* in the effect, and similarly for *bad*), we must alter our mappings. If the mapping for *ho* is changed to *good*, action description (3) cannot contribute to these unwanted consequences any more. But, now action descriptions (1) and (2) may generate unwanted consequences (the condition of (1) is mapped to *good* and the effect to *bad*, and vice versa for (2)). If we also change the mapping for *ta* to *good*, the unwanted consequences generated by action descriptions (1) and (2) also disappear. Our generated pre-interpretation over $D = \{\mathit{block}, \mathit{good}, \mathit{bad}\}$ is:

$0 \to \mathit{block}$
$[\,] \to \mathit{bad}$
$\mathit{ho}(\mathit{block}) \to \mathit{good}$
$\mathit{on}(\mathit{block}, \mathit{block}) \to \mathit{good}$
$[\mathit{good}|\mathit{good}] \to \mathit{bad}$
$[\mathit{good}|\mathit{bad}] \to \mathit{good}$
$s(\mathit{block}) \to \mathit{block}$
$\mathit{em} \to \mathit{bad}$
$\mathit{ta}(\mathit{block}) \to \mathit{good}$
$\mathit{cl}(\mathit{block}) \to \mathit{bad}$
$[\mathit{bad}|\mathit{good}] \to \mathit{good}$
$[\mathit{bad}|\mathit{bad}] \to \mathit{bad}$

The intuition is that we count blocks: terms that identify one block are mapped to *good*, being the abstraction of an *odd* number of blocks, and other terms are mapped to *bad*, being the abstraction of an *even* (including zero) number of blocks. Terms identifying one block are: $\mathit{ho}(\mathit{block})$ (a block being held by the robot), $\mathit{ta}(\mathit{block})$ (a block standing on a table) and $\mathit{on}(\mathit{block}, \mathit{block})$

(a block standing on another block). $cl(block)$ is mapped to *bad* (the block is counted when the other terms are considered). As *em* identifies no block, it is also mapped to *bad*.

If we replace *good* with *odd* and *bad* with *even* in the generated pre-interpretation, we get the following abstract model.

$causes(even, \_, even)$
$ac1\_match(even, even, even)$
$ac1\_match(odd, even, odd)$
$mult\_minus(even, even, even)$
$mult\_minus(odd, even, odd)$
$mult\_subset(even, even, even)$
$mult\_subset(odd, even, odd)$
$append(even, even, even)$
$append(odd, even, odd)$
$action(odd, \_, odd)$
$causes(odd, \_, odd)$
$ac1\_match(even, odd, odd)$
$ac1\_match(odd, odd, even)$
$mult\_minus(even, odd, odd)$
$mult\_minus(odd, odd, even)$
$mult\_subset(even, odd, odd)$
$mult\_subset(odd, odd, even)$
$append(even, odd, odd)$
$append(odd, odd, even)$
$append(even, \_, even)$

It takes 0.35 seconds on a 133 MHz Pentium computer using Sicstus Prolog 3.3 and a naive implementation of a model generator to compute the least abstract model. Note that the interpretation of $[on(s(0), 0), ta(0), cl(s(0)), em]$ is *even* and the interpretation of $[on(s(s(0)), s(0)), on(s(0), 0), ta(0), cl(s(s(0))), em]$ is *odd*. Since any *causes* fact in the abstract model that has *even* as its first argument also has *even* as third argument, we have proved that we have an unsolvable planning problem: $causes(even, \_, odd)$ is false. Our query can therefore never succeed and we have detected an unsolvable planning problem.

The reader should note that we have to balance the following two opposing goals:

- precision of the analysis and
- size of the resulting abstract model.

If each resource is mapped to a different domain element and also each combination of resources is mapped to a different domain element, we get very good precision in the resulting abstract model, but constructing it becomes prohibitively expensive because of the enormous numbers of mappings and therefore also formulas involved. As we are also interested in constructing the simplest model that preserves failure of our planning problem, we want the pre-interpretation with the least number of constants that suffices for this goal. These are clearly two opposing goals that we have to balance. As model generation techniques improve we could then also improve the precision of the proposed analysis method.

The "debugging algorithm" is loosely based on the notion of the preservation of some property/properties (possibly unknown at analysis time) between situations. In classical Blocks World problems, one property that always gets preserved between situations when executing actions, is that the number of blocks in a problem stays the same. In this example, the properties that get preserved are:

1. if our starting situation contains an even number of blocks, all subsequent situations will contain an even number of blocks, regardless of the number and identity of the action description executed, and
2. if our starting situation contains an odd number of blocks, all subsequent situations will contain an odd number of blocks, regardless of the number and identity of the action description executed.

For the described method to work automatically on even more complex examples, we have to:

1. take more than the outer functors of resources into account,
2. be prepared to introduce more than three domain elements in D, and
3. refine our "debugging algorithm".

We end this section with another example from the Blocks World. Some of the refinements just described are needed to detect that this problem is also unsolvable. Consider the problem of generating blocks, putting them on a stack, and checking for a specific block on the stack. The action descriptions defining this problem are:

```
action([ho(V)],put_down(V),[ta(V),cl(V),em]).                         (1b)
action([cl(V),ta(V),em],pick_up(V),[ho(V)]).                          (2b)
action([ho(V),cl(W)],stack(V,W),[on(V,W),cl(V),em]).                  (3b)
action([cl(V),on(V,W),em],unstack(V),[ho(V),cl(W)]).                  (4b)
action([on(X,Y),cl(X),em],generate_block,
       [on(s(X),X),on(X,Y),cl(s(X)),em]).                             (5b)
```

Note the generator of new blocks in action description (5b). The query representing our unsolvable planning problem is:

```
?- causes([on(s(s(0)),s(0)),cl(s(s(0))),em],P,G),
   ac1_match([on(s(0),0)],G,Z).
```

This query "loosely" represents the following problem: is it possible to generate block zero when generating blocks with successive numbers starting from blocks one and two. This is an unsolvable planning problem, but detecting it is again not straightforward. Note that we have an infinite number of cases to consider, but each situation contains only a finite number of blocks. This is obviously not possible to detect that we have an unsolvable problem using resolution based methods. Furthermore, our previous outer functor abstraction is also not powerful enough as we need do differentiate between block zero and the rest of the blocks.

The following mappings however define a pre-interpretation over $D = \{zero, rest\}$, sufficiently precise to prove that our problem is unsolvable.

$$\begin{array}{ll}
0 \rightarrow \mathit{zero} & s(\mathit{zero}) \rightarrow \mathit{rest} \\
s(\mathit{rest}) \rightarrow \mathit{rest} & \mathit{em} \rightarrow \mathit{rest} \\
\mathit{on}(\mathit{zero}, \mathit{zero}) \rightarrow \mathit{zero} & \mathit{on}(\mathit{zero}, \mathit{rest}) \rightarrow \mathit{zero} \\
\mathit{on}(\mathit{rest}, \mathit{zero}) \rightarrow \mathit{zero} & \mathit{on}(\mathit{rest}, \mathit{rest}) \rightarrow \mathit{rest} \\
\mathit{cl}(\mathit{zero}) \rightarrow \mathit{zero} & \mathit{cl}(\mathit{rest}) \rightarrow \mathit{rest} \\
\mathit{ho}(\mathit{zero}) \rightarrow \mathit{zero} & \mathit{ho}(\mathit{rest}) \rightarrow \mathit{rest} \\
\mathit{cl}(\mathit{zero}) \rightarrow \mathit{zero} & \mathit{cl}(\mathit{rest}) \rightarrow \mathit{rest} \\
[\mathit{zero}|\mathit{zero}] \rightarrow \mathit{zero} & [\mathit{zero}|\mathit{rest}] \rightarrow \mathit{zero} \\
[\mathit{rest}|\mathit{zero}] \rightarrow \mathit{zero} & [\mathit{rest}|\mathit{rest}] \rightarrow \mathit{rest}
\end{array}$$

The least abstract model (generated in 0.66 seconds) is:

$$\begin{array}{ll}
\mathit{causes}(\mathit{zero}, \_, \mathit{zero}) & \mathit{causes}(\mathit{rest}, \_, \mathit{rest}) \\
\mathit{ac1\_match}(\mathit{rest}, \mathit{rest}, \mathit{rest}) & \mathit{ac1\_match}(\mathit{rest}, \mathit{zero}, \mathit{zero}) \\
\mathit{ac1\_match}(\mathit{zero}, \mathit{zero}, \mathit{rest}) & \mathit{ac1\_match}(\mathit{zero}, \mathit{zero}, \mathit{zero}) \\
\mathit{mult\_minus}(\mathit{rest}, \mathit{rest}, \mathit{rest}) & \mathit{mult\_minus}(\mathit{zero}, \mathit{rest}, \mathit{zero}) \\
\mathit{mult\_minus}(\mathit{zero}, \mathit{zero}, \mathit{rest}) & \mathit{mult\_minus}(\mathit{zero}, \mathit{zero}, \mathit{zero}) \\
\mathit{mult\_subset}(\mathit{rest}, \mathit{rest}, \mathit{rest}) & \mathit{mult\_subset}(\mathit{rest}, \mathit{zero}, \mathit{zero}) \\
\mathit{mult\_subset}(\mathit{zero}, \mathit{zero}, \mathit{rest}) & \mathit{mult\_subset}(\mathit{zero}, \mathit{zero}, \mathit{zero}) \\
\mathit{append}(\mathit{rest}, \mathit{rest}, \mathit{rest}) & \mathit{append}(\mathit{rest}, \mathit{zero}, \mathit{zero}) \\
\mathit{append}(\mathit{zero}, \mathit{rest}, \mathit{zero}) & \mathit{append}(\mathit{zero}, \mathit{zero}, \mathit{zero}) \\
\mathit{action}(\mathit{rest}, \_, \mathit{rest}) & \mathit{action}(\mathit{zero}, \_, \mathit{zero})
\end{array}$$

$\mathit{causes}(\mathit{rest}, \_, \mathit{zero})$ is false in this model and we have proved that we have an unsolvable planning problem.

5 Automation of the Proposed Method

The proposed method will be further enhanced if it can be automated. In this section we discuss two directions for automation that are being investigated: the first is based on the enumeration of domain elements in D and the second on a refinement of the reasoning of Section 3.

For a method based on enumeration of domain elements in D to be practical, the number of domain elements needs to be small, otherwise the number of mappings that needs to be investigated becomes unmanageable. Although we have not proved it, we have strong evidence from the examples tried so far that very few domain elements are needed to detect failure using the described model-based analysis. The intuition behind this statement is that we are usually trying to capture one property of a problem, e.g. an even or odd number of blocks, preservation of the number of blocks, etc. This is achieved by the mapping of resources and objects in the problem under investigation to two or more domain elements representing positive and negative occurrences of the different properties we are interested in.

The second method is based on a refinement of that presented earlier. In Section 4 we ignored all arguments to resources. A more precise approach might be to identify the objects in the problem description, map them to unique domain elements, and map all other objects that might be generated during execution of action descriptions to one other unique domain element. In our second example, we have blocks $0, s(0)$ and $s(s(0))$ occurring in the query and no other blocks in the action descriptions. Possible mappings that may be generated are:

$$\begin{array}{ll} 0 \rightarrow \mathit{zero} & s(\mathit{zero}) \rightarrow \mathit{one} \\ s(\mathit{one}) \rightarrow \mathit{two} & s(\mathit{two}) \rightarrow \mathit{rest} \\ s(\mathit{rest}) \rightarrow \mathit{rest} & \end{array}$$

These mappings are precise enough to prove that our problem is unsolvable. However, we have redundant domain elements that may complicate the generation of the pre-interpretation. A similar argument as presented earlier may now be used to reason about action descriptions using the mappings we just defined.

Note that all the objections over previous methods, namely loss of argument dependency information as well as inexact counting may be overcome with the proposed pre-interpretation method. The set of terms $\{\mathit{on}(a,b), \mathit{on}(b,a)\}$ may be mapped to two domains elements over $D = \{1,2,3,4\}$ and $\{\mathit{on}(a,a), \mathit{on}(b,b)\}$ to a third and fourth domain element:

$$\begin{array}{ll} \mathit{on}(a,a) \rightarrow 1 & \mathit{on}(a,b) \rightarrow 2 \\ \mathit{on}(b,a) \rightarrow 3 & \mathit{on}(b,b) \rightarrow 4 \end{array}$$

No confusion is therefore possible and no loss of informations occurs. Furthermore, we may count blocks to where necessary as the following mappings illustrate:

$$\begin{array}{ll} [\,] \rightarrow \mathit{zero} & [\mathit{on}] \rightarrow \mathit{one} \\ [\mathit{on}|\mathit{one}] \rightarrow \mathit{two} & [\mathit{on}|\mathit{two}] \rightarrow \mathit{three} \\ [\mathit{on}|\mathit{three}] \rightarrow \mathit{rest} & [\mathit{on}|\mathit{rest}] \rightarrow \mathit{rest} \end{array}$$

In the example we counted up to four—represented by the term *rest*.

Our expectations are that the presented method holds the most promise of all the methods evaluated so far for the detection of unsolvable problems. All unsolvable problems investigated could be detected using the model based analysis and it is difficult to envisage a problem for which this method will not work. However, only further research will show if the presented method can meet our expectation in the long run.

6 Conclusions

The proposed method developed in the previous sections has been restricted to planning problems (and in particular the logic programming approach to deductive planning) for the following reasons:

1. the logic program implementing our planning problem is a definite logic program,
2. the action descriptions are represented by facts,
3. the resources are represented by multisets (lists where the order of elements in the list is not important) and
4. there is a well defined relationship between the resources occurring in the condition and effect of an action description.

These restrictions on general logic programs (and even definite logic programs) make construction of an algorithm for deriving a suitable pre-interpretation to detect failure slightly easier than would be the case for other logic programs. The analysis method is however general and can be used to analyze any definite logic program even though the analysis is not yet completely automatic. The automation of the presented method as well as the extension of the method to definite logic programs in general and Horn clause theorem proving is the subject of current research.

The decision to treat conditions and effects in action descriptions as reversible multisets of resources may effect the precision of the analysis on more complex examples where the direction of actions (from condition to effect) is important. An analysis of other domains will show if this simplification compromises the precision of the analysis or not. An idea may be to develop two analyses: one analyzing the problem forward from the condition in the goal or query and another analyzing the problem backward form the effect in the goal or query. Each analysis may then exploit knowledge about the direction of analysis to obtain further precision. The complexity of the proposed procedure is furthermore dominated by the model generation process. We therefore largely depend on techniques for the fast generation of models for the success of our method.

In [9] planning with abstraction was also investigated. The idea in their work is to find an abstract plan that can be instantiated into a concrete plan for a given planning problem. However, their main aim is to improve the planning efficiency and not to detect unsolvable planning problems. The detection of unsolvable planning problems is mostly a "side effect" of their procedure (as was also the case for Winston [10]).

An investigation of what we achieved with this analysis shows that we have detected infinitely failed computations. In particular, we have detected an infinite number of finitely failed SLD-trees. Furthermore, we introduced into logic programming (and Artificial Intelligence) a semi-automatic model-based analysis method for the detection of unsolvable problems (non-theorems). We further showed that the model-generation method is a viable alternative to "classical" abstract interpretation frameworks giving useful results on interesting problems. The analysis method is elegant in the sense that it is only based on some of the core definitions of the semantics of first order logic, namely *pre-interpretation* and *model*. Stopping short of computing (enumerating) the least Herbrand model of a program, we have an extremely simple framework for detecting non-theorems (as we have shown in previous sections).

References

[1] D.A. de Waal. *Analysis and Transformation of Proof Procedures.* PhD thesis, University of Bristol, October 1994.

[2] D.A. de Waal and M. Thielscher. Solving deductive planning problems using program analysis and transformation. In *Logic Program Synthesis and Transformation.* Springer-Verlag, 1996.

[3] J. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proceedings 1995 International Logic Programming Symposium.* MIT Press, 1995.

[4] J. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[5] S. Hölldobler and J. Schneeberger. A New Deductive Approach to Planning. *New generation Computing*, 8:225–244, 1990.

[6] Steffen Hölldobler and Michael Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(1):99–133, 1995.

[7] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[8] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3):217–242, 1912.

[9] Y. Okubo and M. Haraguchi. Planning with abstraction based on partial predicate mappings. *New Generation Computing*, 12:409–437, 1994.

[10] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, third edition, 1992.