Exploiting Consistency Maintenance for Planning*

Guido Moerkotte and *Holger Müller*

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe
West Germany
Netmail: {*moer,mueller*}@ira.uka.de

1 Introduction

A planning problem mainly consists of a description of an initial world, a set of formulas which declaratively define a goal world and a set of operator specifications which can be used to derive new worlds from given worlds. The goal of the planning procedure is to find a sequence of operators such that if they are applied to the initial world in succession, they result in a goal world in which all the formulas of the goal description hold.

Deductive databases can be described by a triple consisting of a set of facts, a set of rules, and a set of constraints that each database state has to obey. Deductive databases will be used to model worlds in a planning domain. Further, we assume the existence of a consistency maintenance procedure which allows to find for an inconsistent database state consistent ones.

The exploitation of automatic consistency maintenance is the main issue of the paper. Hence, it will be described first. A simple example will motivate the usefulness of consistency maintenance for planning. Then, a planning procedure and some variants are specified which build upon consistency maintenance. This planner mainly consists of two layers, the deductive database and a specialized procedure devoted to planning. We call this a hybrid or heterogeneous approach. The basic planning procedure as well as some more advanced ones will be specified and some theoretical and empirical results will be given.

While this planner outperforms existing ones in both, expressiveness and performance, there still exist some problems which have to be tackled in order to improve its performance. This will lead to a new, homogeneous planner. The key idea is to axiomatize the planning procedure and use consistency maintenance to derive descriptions of possible plans.

*This work was supported by Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314 "Artificial Intelligence and Knowledge-Based Systems", Project D5.

2 Deductive Databases

This section review the main results of [11].

2.1 Preliminaries

To begin with, we quickly review the subset of first-order predicate calculus that we will use throughout this paper. A **signature** Σ is a triple $(K_\Sigma, Pr_\Sigma, \alpha_\Sigma)$ where K_Σ is a set of constants, Pr_Σ is a set of predicates, and α_Σ is a mapping $\alpha_\Sigma : Pr_\Sigma \longrightarrow \mathbf{N}$ called arity. Variables are denoted by x,y,z,... possibly with subscript. A **term** is a variable symbol or a constant symbol. The set of all terms is denoted by T_Σ. $Subst_\Sigma = \mathcal{P}(\{x/t \mid x \text{ variable}, t \in T_\Sigma\})$ is the set of **substitutions**. The set of **atoms** is defined as $At_\Sigma := \{p(t_1...t_n) \mid t_i \in T_\Sigma, \alpha_\Sigma(p) = n, p \in Pr_\Sigma\}$. An atom, not containing any variable is called a **ground atom** or simply a **fact**. Fak_Σ denotes the set of all facts. Any atom is a **literal**. If l is a literal then $\neg l$ is a literal. The set of all literals is denoted by Lit_Σ, the set of all ground literals by $GLit_\Sigma$. If $l_1, ..., l_n$ are literals and l_{n+1} is an atom then $l_1, \ldots, l_n \Longrightarrow l_{n+1}$ is a **rule**. All atoms are **formulas**. If f_1 and f_2 are formulas then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$, and $f_1 \Longrightarrow f_2$ are formulas. Further, if x is a variable and f is a formula then $\forall\, x\ f$ and $\exists\, x\ f$ are formulas. The set of all formulas is denoted by For_Σ.

A **database** is a triple $DB := \langle DB^a, DB^r, DB^c \rangle$ where DB^a is a set of facts, DB^r is a set of stratified rules, and DB^c is a set of closed formulas called consistency constraints. A predicate of Pr_Σ which occurs in a head of a rule in DB^r is called a **derived predicate**, otherwise the predicate is called a **base predicate**. We assume that DB^a only consists of facts of base predicates. The set of all facts which are eligible for DB^a is denoted by Fak_Σ^{base}. The **extension** of a database is defined as

$$M(DB) := \{a \mid a \ is \ a \ fact, DB^a \cup DB^r \models a\}.$$

The **completion** of a database DB (see [13]) is given by

$$C(DB) := M(DB) \cup \{\neg a \mid \text{ a is a fact}, DB^a \cup DB^r \not\models a\}.$$

We abbreviate $C(DB) \models$ by $DB \models$. A database is called **consistent** if $C(DB) \cup DB^c$ is consistent. Deviating from the set-up in classical logic, but in accordance with its usage in deductive database theory we define a set F of formulas to be consistent if there is a model M of F where every element of M is named by exactly one constant symbol. Thus consistency of $C(DB) \cup DB^c$ is equivalent to $DB \models \varphi$ for each $\varphi \in DB^c$. We make the reasonable assumption that $DB^r \cup DB^c$ is consistent.

A **transaction** is a finite sequence $TA := \alpha_1(a_1), \ldots, \alpha_n(a_n)$ where $\alpha_i \in \{add, del\}$ are **instructions** and a_i are ground atoms. Further, a mapping Γ from the set of all transactions to a subset of all ground literals is defined as $\Gamma(TA) := \{a | del(a) \in TA\} \cup \{\neg a | add(a) \in TA\}$. If not stated otherwise, only transactions with $\Gamma(TA)$ consistent will be considered.

2.1.1 Symptoms, Causes, and Repairs

Now, the main definitions for symptoms, causes, and repairs follow. For each of these definitions two different cases are considered. First, the notions are defined for the case of

a single consistency constraint, and subsequently for the case of the entire set DB^c. The former will be referred to as potential symptoms (causes, repairs), the latter as definite symptoms (causes, repairs). For the definitions two mappings are needed. The first one ($\circ_S$) will be used to modify the completion $C(DB)$ of a database whereas the second ($\circ_C$) is used to denote modifications to the fact base DB^a.

Definition 2.1 *If $A, B \subseteq Lit_\Sigma$ then $\circ_S, \circ_C : \mathcal{P}(Lit_\Sigma) \times \mathcal{P}(Lit_\Sigma) \longrightarrow \mathcal{P}(Lit_\Sigma)$ are defined as follows:*

1. $A \circ_S B := (A \setminus B) \cup \overline{B}$
2. $A \circ_C B := (A \setminus \{b | b \in B, b \in At_\Sigma\}) \cup \{b | \neg b \in B, b \in At_\Sigma\}$

where $\overline{B}$ for a set of literals is defined as $\overline{B} := \{a | \neg a \in B, \text{a is an atom}\} \cup \{\neg a | a \in B, \text{a is an atom}\}$, and $\mathcal{P}$ denotes the powerset operator. Further define $DB \circ_C X$ for a set X of ground literals as $\langle DB^a \circ_C \overline{X}, DB^r, DB^c \rangle$

Let $a, b, c, d, e, \ldots$ be facts. Consider the following database $DB = (\{a, b\}, \{a \Longrightarrow c\}, \emptyset)$. Then $C(DB) = \{a, b, c, \neg d, \neg e \ldots\}$. If we now wish the extension of DB to exclude a but include d this can be denoted by $C(DB) \circ_S \{a, \neg d\}$ which results in $\{\neg a, b, c, d, \neg e, \ldots\}$. That d is added to the fact base and a deleted from it can be denoted by $DB^a \circ_C \{a, \neg d\}$. The completion of the thus modified database is slightly different, namely $\{\neg a, b, \neg c, d, \neg e, \ldots\}$.

The example demonstrates that the completion of a database can be seen as the contents of the database as it appears to the user. Hence, it makes sense from a pragmatic standpoint as well to use the completion to define the consistency of the database. Consequently, the analysis of a consistency violation is first performed at the level of the completion. This yields symptoms which are allowed to be derived facts. In a second step causes are derived from these symptoms, which directly concern DB^a.

One problem arises if several consistency constraints are presented. It is quite possible that the "repair" of one consistency constraint may violate another. Therefore we distinguish between a potential symptom which is concerned with a single consistency constraint, and a definite symptom which takes into consideration the entire set of consistency constraints.

Definition 2.2 (symptom) *For a database $DB = \langle DB^a, DB^r, DB^c \rangle$, and $\varphi \in DB^c$ with $C(DB) \cup \{\varphi\}$ inconsistent, a subset $S \subseteq Lit_\Sigma$ of ground literals is called*

1. *(potential) symptom $:\prec\succ$ $(C(DB) \circ_S S) \cup \{\varphi\}$ consistent and S minimal*
2. *definite symptom $:\prec\succ$ $(C(DB) \circ_S S) \cup DB^c$ consistent and S minimal*

In other words, a symptom is a set of literals that must be applied by $\circ_S$ in order to restore consistency.

Having detected the symptoms does not by itself help to alleviate the consistency violation, since the facts in the completion need not necessarily be present in the database. This is especially true for negated facts. On the other hand, a repair can certainly be applied only to the set DB^a of stored facts. Hence, we need a notion that relates repairs to such facts. This notion, referred to as cause, should be differentiated in a way similar to symptoms.

Definition 2.3 (cause) *Be $C \subseteq Lit_\Sigma$, $DB = (DB^a, DB^r, DB^c)$ a database, and $\varphi \in DB^c$. If $C(DB) \cup \{\varphi\}$ is inconsistent then C is called*

1. *(potential) cause* $:\prec\succ C(DB \circ_C C) \cup \{\varphi\}$ *consistent and C minimal*
2. *definite cause* $:\prec\succ C(DB \circ_C C) \cup DB^c$ *consistent and C minimal*

Hence, a cause is a set of literals that must be eliminated by applying $\circ_C$ in order to regain consistency.

Clearly, one of our subsequent objectives must be to establish a connection between symptoms and causes. Ultimately we wish to provide the user with a transaction, i.e., a sequence of operations which resolve the inconsistency. This kind of transaction will be called repair.

Definition 2.4 (repair) *Be TA a transaction, $DB = (DB^a, DB^r, DB^c)$ a database and $\varphi \in DB^c$ a consistency constraint. If $C(DB) \cup \{\varphi\}$ is inconsistent then TA is called*

1. *(potential) repair* $:\prec\succ C(TA(DB)) \cup \{\varphi\}$ *consistent and TA minimal*
2. *definite repair* $:\prec\succ C(TA(DB)) \cup DB^c$ *consistent und TA minimal*

Obviously, a transaction TA is a repair iff $\Gamma(TA)$ is a cause, e.g., if $\{del(p(a)), add(q(b))\}$ is a repair then, taking into account the definition of $\circ_C$, $\{p(a), \neg q(b)\}$ clearly is a cause, and vice versa. Thus, the paper will concentrate on the steps of extracting symptoms from the inconsistency, i.e., violated consistency constraints, and on the steps of generating the causes from the symptoms.

2.2 Extracting Symptoms from Inconsistency

In this chapter the step of extracting symptoms from inconsistencies is taken. For this, the violated consistency constraint, more specifically instances of it, are analyzed.

For any formula f in prenex normal form with its matrix in disjunctive normal form, which is inconsistent with $C(DB)$, we define a set $Symp(f)$ of sets of literals. Each $S \in Symp(f)$ is a symptom for the inconsistency of f and $Symp(f)$ is intended to contain all symptoms.

Definition 2.5 (Symp(f))

- *If* $f = f_1 \wedge \ldots \wedge f_n$ *with literals* f_i *then*
  $\text{Symp}(f) = \{\{f_j | C(DB) \not\models f_j, 1 \leq j \leq n\}\}$;
- *If* $f = f_1 \vee \ldots \vee f_k$ *then*
  $\text{Symp}(f) = \bigcup\{\text{Symp}(f_i) | C(DB) \not\models f_i, 1 \leq i \leq k\}$;
- *If* $f = \forall x f_0$ *then*
  $\text{Symp}(f) = \{s_1 \cup \cdots \cup s_k \mid s_i \in \text{Symp}(f_0[x \leftarrow c_i])\}$
  where $\{c_1, \ldots, c_k\} = \{c \in K_\Sigma \mid C(DB) \not\models f_0[x \leftarrow c]\}$
- *If* $f = \exists x f_0$ *then*
  $\text{Symp}(f) = \bigcup\{\text{Symp}(f_0[x \leftarrow c] \mid c \in K_\Sigma\}$.

Of course, considering all constants $c \in K_\Sigma$ in the case of existential quantifiers is intractable. The problem of "guessing" the right constants is beyond the scope of this paper and described in [12] and [9].

Theorem 2.6 *If $C(DB) \circ_S W \models f$, then there is $S \in \mathrm{Symp}(f)$ with $S \subseteq W$.*

Note that the application of MIN to $Symp(f)$ guarantees the result to contain exactly the potential symptoms.

2.3 Generating Causes from Symptoms

Generating causes from symptoms is related to the view update problem in that updates at the intentional level must be compiled to the extensional level. The following procedure resembles [15]. The central notion of this chapter which lays the foundation of all essential results is the notion of **derivation tree**.

Definition 2.7 (Derivation Tree DT_∞) *For a set of facts A, a set of stratified rules D, and a ground literal $l \in GLit_\Sigma$ we define the derivation tree $DT_\infty(l, A, D)$. $DT_\infty(l, A, D)$ is an and-or-tree where the nodes are labled with ground literals $b \in GLit_\Sigma$ or pairs (r, σ) for a rule $r \in DB^r$ and a substitution $\sigma \in Subst_\Sigma$. Further the following conditions must hold:*

1. *The root is labeled by l.*
2. *For each node N labeled with a derived literal b and for each pair (r, σ) s.t.*
   - $r = l_{n+1} \Longleftarrow l_1, \ldots, l_n$,
   - $l_{n+1}\sigma = |b|$, *and*
   - $r\sigma$ *closed,*

   N is followed by a node N' labeled with (r, σ).
   - *If l is a positive literal, N is an or-node and N' is an and-node which has as successor nodes exactly the roots of $DT_\infty(l_1\sigma, A, D), \ldots, DT_\infty(l_n\sigma, A, D)$.*
   - *If l is a negative literal, N is an and-node and N' is an or-node which has as successor nodes exactly the roots of $DT_\infty(\overline{l_1}\sigma, A, D), \ldots, DT_\infty(\overline{l_n}\sigma, A, D)$.*
3. *For each node N labeled with a literal b of a base predicate, if b is positive and $b \in DB^a$ holds, N is an or-node and N has exactly one more and-successor node labeled by b.*

   If b is a negative literal and $\overline{b} \notin DB^a$ holds, N is an and-node and N has exactly one or-successor node labeled by b.

Derivation trees represent all possibilities to derive the literal l from $A \cup D$ using input resolution and negation-as-failure as the only inference rule. In this respect derivation trees are similar to the SLDNF-trees as defined in, e.g., [8]. The differences are, that in derivation trees labels to or-nodes are atoms, while the corresponding nodes in SLDNF-trees are labeled by conjunctive goals and furthermore that and-nodes are not explicitly represented in SLDNF-trees.

Example 2.8
Signature: $Pr_\Sigma = \{a, b, c, d, e, f\}$, $\forall p \in Pr_\Sigma : \alpha_\sigma(p) = 0$
Facts: $DB^a = \{a, b, d\}$
Rules: $DB^r = \{e \Leftarrow c, d; f \Leftarrow a, b; f \Leftarrow d, \neg e; e \Leftarrow c, \neg a; e \Leftarrow c, \neg b\}$
The derivation tree for e is shown in Figure 1. Circles correspond to or-nodes whereas rectangles correspond to and-nodes.

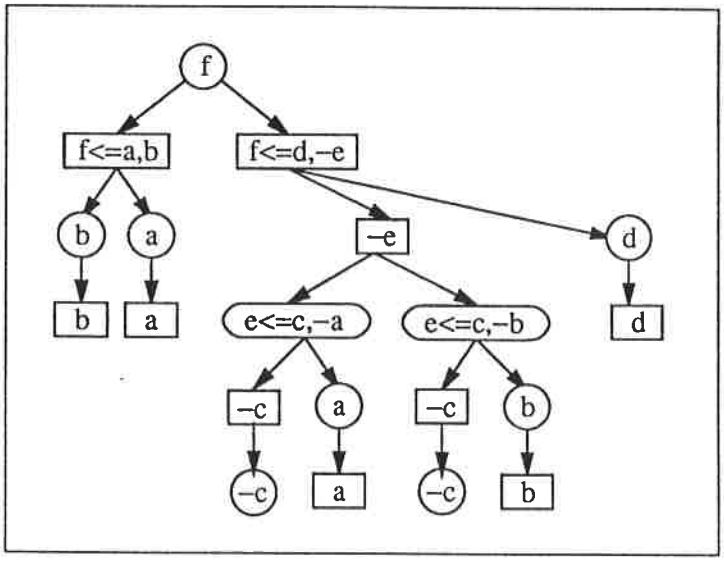

Figure 1: Derivation tree $DT\ (e)$ of the example

Though it is intuitively clear how to translate derivation trees into SLD-trees and vice versa the technical details are cumbersome. The equivalence of both concepts can be established via the fixpoint characterization (for further details see [11]). Derivations in derivation trees will play the same role as successful branches in SLD-trees.

Definition 2.9 (Derivation) *Let* $DB = \langle DB^a, DB^r, DB^c \rangle$ *be a database. For a ground literal* $l \in GLit_\Sigma$ *a derivation* $DE(l)$ *of* l *over* $DT_\infty(l, A, D)$ *is defined as a finite subtree of* $DT_\infty(l, A, D)$ *where the following conditions hold:*

1. *The root of* $DE(l)$ *is the same as the root of* $DT_\infty(l, A, D)$.
2. $DE(l)$ *contains for each or-node exactly one successor node.*
3. $DE(l)$ *contains for each and-node all the successor nodes of the corresponding and-node in* $DT_\infty(l, A, D)$.
4. *Every leaf* N *in* $DE(l)$ *is a node labeled by a ground base literal* l'. *If* l' *is positive,* N *is an and-node, otherwise* N *is an or-node.*

Lemma 2.10 *Let* $DB = \langle DB^a, DB^r, DB^c \rangle$ *be a database and* l *an arbitrary ground literal. Then*

$$l \in C(DB) \prec\succ \text{there exists a derivation of } l \text{ in } DT_\infty(l, DB^a, DB^r).$$

In dealing with derivation trees we encounter the problem, that even when a (finite) derivation exists, the derivation tree itself may be infinite:

Example 2.11 *Let* $DB^a = \emptyset$, $DB^r = \{b \Leftarrow a; a \Leftarrow b;\}$. $DT_\infty(b, DB^a, DB^r)$ *is an infinite derivation tree.*

Since we deal only with ground formulas there is an easy solution to this problem. We call a derivation **repetition free** if no branch contains two nodes labeled by the same literal.

Definition 2.12 (repetition free derivation tree DT**)** *The repetition free derivation tree* $DT(l, A, D)$ *is the subtree of* $DT_\infty(l, A, D)$ *obtained by applying iteratively the following operations:*

1. *If two or-nodes* N_1 *and* N_2 *are marked by the same label and* N_2 *is a descendent of* N_1 *, delete* N_2 *and all nodes following* N_2.
2. *If at least one successor node of the and-node* N *has been deleted, delete* N *also.*
3. *If all successor nodes of the or-node* N *have been deleted, then delete* N *also.*

For the rest of the paper we abbreviate $DT(l, DB^a, DB^r)$ by $DT(l)$ and $DT_\infty(l, DB^a, DB^r)$ by $DT_\infty(l)$.

Lemma 2.13 *Let* $DB = \langle DB^a, DB^r, DB^c \rangle$ *be a database,* $l \in GLit_\Sigma$.

1. *there exists a derivation of* l *from* DB $\prec\succ$
   there exists a repetition free derivation of l *from* DB
2. *there exists a repetition free derivation of* l *in* $DT_\infty(l)$ $\prec\succ$
   there exists a derivation of l *in* $DT(l)$

Sometimes when l is not derivable from DB, $DT(l)$ can be the empty tree. Since our set-up does not include function symbols repetition-free derivation trees are always finite. In the general case that need not be true, but 2.13 continues to hold.

We are now ready for the definition of $causes(l)$ or more precisely $causes(l, DB)$. The idea is that $causes(l)$ is a collection of sets X of ground literals, such that

$$DB \circ_C X \not\models l$$

In fact $causes(a)$ will contain exactly the sets X which are minimal with respect to the stated non-derivability condition. Minimality will be achieved by respected use of the minimum operator MIN, which for any collection A of sets is defined by:

$$MIN(A) = \{X : X \text{ is a minimal set in A}\}$$

Definition 2.14 (*causes*) *Let* DB *be a database,* $l \in GLit_\Sigma$. *If* $DB \not\models l$, *then* $causes(l) = \emptyset$. *Now assume* $DB \models l$. *For all nodes* N *in the derivation tree* $DT(l)$ $causes(N)$ *is inductively defined by*

1. *If N is an leave labeled by l' with no successor:*

   if N is an and-node and l' is a positive literal or N is an and-node and l is negative,

   - *then* $causes(N) = \{\{l'\}\}$.
   - *else* $causes(N) = \emptyset$

2. *If N is an and-node with successor nodes $N_1, \ldots, N_k$, then* $causes(N) = MIN(causes(N_1) \cup \cdots \cup causes(N_k))$.

3. *If N is an or-node with successor nodes $N_1, \ldots, N_k$, then* $causes(N) = MIN(\{C_1 \cup \cdots \cup C_k \mid C_i \in causes(N_i)\})$

Finally $causes(l) = causes(R)$ *where R is the root of* $DT(l)$.

We illustrate the definition by the example 2.8. Clearly $DB \models e$. The derivation tree was shown in Figure 1. Remember that the and-nodes are surrounded by squares and the or-nodes by circles. We have:

- for example, for the and–node $\neg e$: $\{\{a, \neg c\}, \{b, \neg c\}, \{d\}\}$
- and for the root f the following causes: $\{\{a, \neg c\}, \{a, d\}, \{b, \neg c\}, \{b, d\}\}$

We state the following elementary properties of $causes(N)$.

Lemma 2.15 *1. Let N be an or-node in $DT(a)$ with successor nodes $N_1, \ldots, N_k$ then*

- *for all $X \in causes(N)$ and all j, $1 \leq j \leq k$ there is a set $Y_j \in causes(N_j)$ with $Y_j \subseteq X$*
- *for all $Y_1 \in causes(N_1), \ldots, Y_k \in causes(N_k)$ either there are $Y_1' \in causes(N_1), \ldots, Y_k' \in causes(N_k)$ with $Y_1' \cup \ldots \cup Y_k' \subset Y_1 \cup \ldots \cup Y_k$ or $Y_1 \cup \ldots \cup Y_k \in causes(N)$.*

2. Let N be an and-node in $DT(a)$ with successor nodes $N_1, \ldots, N_k$ then

- *for all $X \in causes(N)$ there is an index j, $1 \leq j \leq k$ with $X \in causes(N_j)$*
- *for all j, $1 \leq j \leq k$ and every $Y \in causes(N_j)$ there is some j_0, $1 \leq j_0 \leq k$ and some $Y_0 \in causes(N_{j_0})$ with $Y_0 \subset Y$ or $Y \in causes(N)$.*

Theorem 2.16 (Completeness of *causes*) *Let DB be a database, $l \in GLit_\Sigma$, such that $DB \models l$ and X a minimal set of ground base literals such that $(DB \circ_C X) \not\models l$. Then $X \in causes(l)$.*

Definition 2.17 (Causes) *For a database and a set $S = \{l_1, \ldots, l_n\}$ of ground literals we define* CAUSES$(S) = MIN\{g_1 \cup \ldots \cup g_n |$ *for all* $1 \leq i \leq n$, $g_i \in causes(l_i)\}$.

Now, the main theorem of this section can be stated.

Theorem 2.18 *(symptom $\rightsquigarrow$ cause) Be Σ a signature, $DB = (DB^a, DB^r, DB^c)$ a database, $\varphi \in DB^c$ a consistency constraint, $C(DB) \cup \{\varphi\}$ inconsistent, and $S \subseteq Lit_\Sigma$ a symptom. Then all causes C with $C(DB \circ_C C) = C(DB) \circ_S S$ are contained in* CAUSES(S).

Note that there might exist a $C \in \text{CAUSES}(S)$ such that $C(DB \circ_C C) \neq C(DB) \circ_S S$, and that there may be no g such that $C(DB \circ_C C) = C(DB) \circ_S S$. Consider the following example:

$$DB^a = \{a\},\ DB^r = \{a \Longrightarrow b\},\ S = \{\neg b\}$$

then

$$C(DB \circ_C C) \neq C(DB) \circ_S S$$

for all C.

Further note that $DB \circ_C C$ must not be consistent with all consistency constraints but only with the one under consideration (potential cause vs. definite cause). The problem of extracting causes from symptoms is solved by iterating the indicated process as stated in the next subsection.

2.4 Iterative Construction of Definite Causes

Up to now, we have only been concerned with the extraction of potential causes. Note the following connection between potential and definite causes:

- C is a definite cause $\succ$ there exists a potential cause C' s.t. $C' \subseteq C$.

In other words, usually the same literal will cause only the violation of one constraint. If several constraints are violated this will be due to different causes. Consequently, a definite cause will usually be a combination of potential causes, limited however to those combinations whose associated repair will restore all violated constraints. The following procedure will iteratively compute such a "correct" combination (or set) from the potential causes.

Algorithm 2.19 *This algorithm generates the set of definite causes for a given inconsistency. pc^i denote a set of literals for a natural number i, PC is the set of potential causes, and DC is the set of definite causes. The upper script i of pc^i denotes the level of the cause, i.e., the number of enhancements, which were made to transform the previous potential causes of pc^i to a definite one.*

(1) ***procedure*** $\text{CAUSES}_\infty(DB)$
(2) ***input:***
(3) $DB = (DB^a, DB^r, DB^c)$: *(inconsistent) database state*
(4) ***init*** $PC := \{pc^0 \mid pc^0$ *is a (potential) cause for* $DB\}$,
(5) $DC := \emptyset$,
(6) ***while*** *there exists some* $pc^i \in PC$ ***do***
(7) $PC := PC \setminus \{pc^i\}$
(8) ***if*** $\Gamma^{-1}(pc^i)(DB)$ *is consistent*
(9) ***then*** $DC := DC \cup \{pc^i\}$
(10) ***else***
(11) ***if*** pc^i *is free of contradictions*
(12) ***then***
(13) *Let* $pc_1, \ldots, pc_n$ *be all potential causes for* $\Gamma^{-1}(pc^i)(DB)$.
(14) $pc_k^{i+1} := pc^i \cup pc_k$, *for all* $k = 1, \ldots, n$.
(15) $PC := PC \cup \{pc_k^{i+1} \mid 1 \leq k \leq n\}$.
(16) ***return*** *definite causes* DC

Obviously, the algorithm computes the set of all definite repairs, if Γ^{-1} is applied on the resulting causes. This set may be very large and not all repairs are of equal value. Hence, heuristics have to be developed to restrain the set of generated repairs and impose an order on the remaining ones such that the system is able to present the "top ten" repairs for a given violation. This issue is beyond the scope of this paper and is treated in full detail in [12, 9]. A simple solution for this problem is the restriction of the iteration level i. Instead of computing all definite causes, one only considers the causes of the first i levels, i.e., the (potential) causes pc^k where $k \leq i$. The set of these causes is denoted by CAUSES_i.

3 Consistency Driven Planning

3.1 Introduction and Motivation

We describe an approach to planning based on deduction. STRIPS [5] and the situation calculus [6] show two extremes of using deduction within a planner. In STRIPS, deduction is only used for checking preconditions of actions, while situation calculus does *everything* by deduction using a theorem prover. The latter does not seem appropriate when one is interested in building usable planners, since small toy problems already overwhelm state-of-the-art provers with inferences [2].

The approach we present in this section lies between these two extremes. On one hand, we will model worlds W by deductive database states. World transitions can then be modeled by transaction leading from one database state to the next. On the other hand, we leave the actual search for a plan to a specialized algorithm which builds upon the primitives provided by a deductive database system mainly executing transactions and finding repairs.

Since it is common practice to denote worlds by W and not by DB, we define a world to be a triple $W = \langle W^a, W^r, W^c \rangle$, where the semantics is the one defined for deductive databases.

The planning problem can now easily be stated as follows: Given an initial world W_0 and a goal G, find a sequence of operations—called plan—that leads from W_0 to W_G with $W_G \models G$. In our approach the goal can be any set of first-order formulas. This generalizes the possible language to describe goals found in the literature, since there, only a conjunction of facts is allowed to state a goal. An operator is always an instance of an operator specification. An operator specification is defined as follows:

Definition 3.1 (operator specifications) *An* operator specification *is defined as*

declare $opsym(x_1, \ldots, x_n)$
Range $F_{range} \subset \text{For}_\Sigma$
Pre $F_{pre} \subset \text{For}_\Sigma$
Post $L_{post} \subset \text{Lit}_\Sigma$

where opsym is the name of the operator and the parameters $x_1, \ldots, x_n$ *are variables. All predicates of* F_{range} *have to be protected (i.e., it must be assured that their extension never changes) and* L_{post} *must not contain any derived predicate. The formulas*

of F_{range}, F_{pre}, and L_{post} may contain only $x_1,\ldots,x_n$ as free variables. The formulas $\exists x_1,\ldots,\exists x_n \bigwedge_{f\in F_{range}} f$ and $\bigwedge_{f\in F_{pre}} f\sigma$ where σ is a ground substitution for $x_1,\ldots,x_n$ have to be range restricted.

Then, an **operator** is a head of an operator specification where all parameters are replaced by constants. *range(op)*, *pre(op)* and *post(op)* refer to the fully instantiated sets of F_{range}, F_{pre}, and L_{post}, respectively. An operator *op* is a **valid instantiation** of an operator specification with respect to a world W if $W \models range(op)$. Note that this operator definition again generalizes the common operator definitions in that the language used for specifying preconditions is more powerful.

A **plan** is a sequence of operators. The application of a plan $[op_1,\ldots,op_n]$ on a world W is defined by the application of the concatenated operators $op_n \circ \cdots \circ op_1$ on W. Each plan $[op_1,\ldots,op_n]$ defines a unique sequence of possible worlds $W_1,\ldots,W_n$:

- $W_i^a := W_{i-1}^a \circ_C post(op_i)$

whereas $W_i^r = W^r = W_0^r$ and $W_i^c = W^c = W_0^c$ remain unchanged. An operator op_i is called a **producer** of a literal l for a world W_j if $l \in post(op_i)$, $W_{i-1} \not\models l$, and $W_k \models l$ for each $k = i,\ldots,j$. In the case of $\bar{l} \in post(op)$ and $W_{i-1} \models l$ the operator op_i is a **destroyer** of l.

We are now ready to state the planning problem formally: A **planning problem** is a triple $\langle W_0, OP, G\rangle$, where W_0 is a consistent initial world, OP a set of operator specifications, and G is a set of closed, range-restricted formulas called the **goal**. For uniformity reasons, we assume that a goal G implicitly introduces an operator *goal*, where $pre(goal) := G$ and *post(goal)* is empty.

A plan is called a **solution** for a planning problem, iff

1. $op_n = goal()$,
2. $\forall i \in \{1,\ldots,n\} : Comp(W_{i-1}) \models pre(op_i)$, and
3. $\forall i \in \{1,\ldots,n\} : W_i$ is consistent.

In terms of planning, this means that before inserting an operator into an operator sequence two things have to be verified: (i) the preconditions of the operator must be satisfied, and (ii) the resulting world must be consistent. An operator which violates one of these conditions is called **critical**.

Let us now demonstrate these definitions by means of a simple example. This will also motivate why we use the consistency maintenance mechanism introduced in the last section for planning purposes. The planning problem considered is the conjunctive goal problem or Sussman Anomaly. The planning domain is the blocks world. The initial and the goal world are depicted in figure 2.

Mainly, W_0 is given by the facts $\{on(a,table)$, $on(c,a)$, $on(b,table)\}$. The rules and constraints W_0^r and W_0^c verbally read as follows:

support-rule-1:	an object which stands on another one is supported by that object
support-rule-2:	every object supports all objects that stand on it

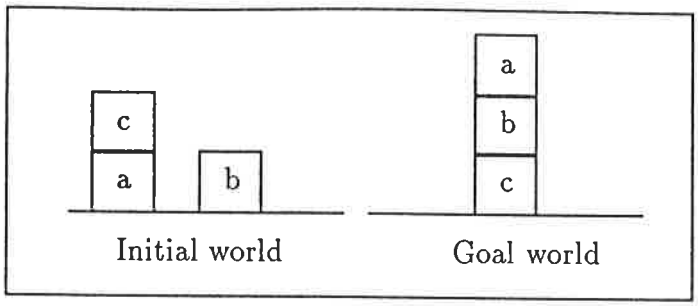

Figure 2: The conjunctive goal problem

constr-1: everything can be at one place only
constr-2: only one block can be on another
constr-3: every block is supported by the table
constr-4: only blocks can stand on other objects

A complete axiomatization is presented in figure 3. The goal G is $\{on(a,b),\ on(b,c)\}$.

W_0^r:

support-rule-1 : $\forall x_1, x_2 :\quad on(x_1,x_2) \Rightarrow supported\text{-}by(x_1,x_2)$

support-rule-2 : $\forall x_1, x_2, x_3 :\quad on(x_1,x_2) \wedge supported\text{-}by(x_2,x_3) \Rightarrow supported\text{-}by(x_1,x_3)$

W_0^c:

constr-1 : $\forall x_1, x_2, x_3 :\quad on(x_1,x_2) \wedge on(x_1,x_3) \Rightarrow x_2 = x_3$

constr-2 : $\forall x_1, x_2, x_3 :\quad is(x_1,block) \wedge on(x_2,x_1) \wedge on(x_3,x_1) \Rightarrow x_2 = x_3$

constr-3 : $\forall x_1 :\quad is(x_1,block) \Rightarrow supported\text{-}by(x_1,table)$

constr-4 : $\forall x_1, x_2 :\quad on(x_1,x_2) \Rightarrow is(x_1,block)$

Operator:

declare move(x_1,x_2,x_3)

range $is(x_1,block),\ x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3$
$is(x_2,block) \vee x_2 = table,$
$is(x_3,block) \vee x_3 = table,$

prec $on(x_1,x_2),\ \forall x_4\ \neg on(x_4,x_1)$

post $\neg on(x_1,x_2),\ on(x_1,x_3)$

Figure 3: Formalization of the blocks world

In order to motivate the approach of **goal completion**, let us first demonstrate how STRIPS or STRIPS–like planners would work on this problem. Since the language of STRIPS is quite restrictive—no formulas, only atoms are allowed in the preconditions—the above operator specification of *move* is not a valid STRIPS operator. This results in a slightly different modeling of the blocks world domain, since no negation is allowed. More specifically, an additional predicate *clear* is needed which states that for *clear(x)* there is no block on x. Since there are also no deductive rules in STRIPS, the operators have to take care of the validity of this predicate.

Hence, for STRIPS, several different operator specifications are required depending on the different actions they take, e.g.:

1. moving a block from a block onto a block:

   declare move1(x, y, z)
   prec $x \neq table, y \neq table, z \neq table$
   $on(x, y), clear(x), clear(z),$
   post $\neg clear(z), \neg on(x, y)$
   $clear(y), on(x, z)$

2. moving a block from a block onto the table:

   declare move2$(x, y, table)$
   prec $x \neq table, y \neq table, x \neq y$
   $on(x, y), clear(x)$
   post $\neg on(x, y)$
   $clear(y), on(x, table)$

3. moving a block from the desk onto a block:

   declare move3$(x, table, z)$
   prec $x \neq table, z \neq table, x \neq z$
   $on(x, table), clear(x)$
   post $\neg clear(y), \neg on(x, table)$
   $on(x, z)$

This distinction can be avoided since we allow derived predicates and general formulas—which obviously subsume negative literals.
The strategy, the STRIPS planner uses is adopted from the General Problem Solver ([3]):

Algorithm 3.2 (GPS–Planner)

```
(1)  procedure GPS(W,op)
(2)  input:
(3)  W: consistent world
(4)  op: an operator
(5)  init plan := [];
(6)  while op is critical w.r.t. W do
(7)    choose an unsatisfied subgoal g of pre(op);
(8)    choose an operator op': g ∈ post(op');
(9)    plan' := GPS(W, op');
(10)   plan := plan ∘ plan' ∘ [op'];
(11)   W := plan' ∘ [op'](W);
(12) return(plan);
```

The planner starts with the original world W_0 and a special operator *goal* whose precondition is the goal G and whose postcondition is empty . First, *plan* is initialized with the

empty plan; some unsatisfiable subgoal in $pre(op)$[1] is chosen and an operator op' is chosen whose postcondition just satisfies this goal. Then, within a recursive call, a subplan, which fulfills the precondition of op', is constructed. The resulting plan is connected to the current plan and the loop is entered again, until no more unsatisfied subgoals of op can be found.

Let us demonstrate this algorithm for the conjunctive goal problem of Figure 2 where we abstract from the facts stating that something is a block or a table. We simply concentrate on the extensions of the *on* predicate. There are two subgoals we can start with. Namely, *on(a,b)* and *on(b,c)*:

on(a,b) The only operator op' which could lead to an optimal solution is *move3(a,table,b)*. This results in the recursive call *GPS(W,move3(a,table,b))*.

Out of the preconditions *on(a,table)* and *clear(a)*, the latter is unsatisfied. Hence, this operator is critical. An operator having *clear(a)* in its postcondition is *move2(c,a, table)*. The resulting recursive call is *GPS(W,move2(c,a,table))*. The preconditions of this operator are fulfilled. This results in unwinding the recursive calls.

At the upper GPS level we now have *plan = [move2(c,a,table), move3(a,table,b)]* and $W = \{on(c,table),\ on(a,b), on(b,table)\}$.

There exists still the unsatisfied precondition *on(b,c)* of the artificial operator *goal*. The applicable operator is *move3(b,table,c)*. The recursive call *GPS(W,move3(b, table,c))* results in *plan' = [move2(a,b,table)]*.

Hence, at the outer level we have *plan = [move2(c,a,table), move3(a,table,b), move2(a, b,table), move3(b,table,c)]* and $W = \{on(b,c),\ on(b,table),\ on(a,table)\}$. Hence, there still is the unsatisfied subgoal *on(a,b)*. The applicable operator is *move3(a,table, b)* whose precondition is fulfilled.

Summing up, the resulting plan is *[move2(c,a,table), move3(a,table,b), move2(a, b,table), move3(b,table,c), move3(a,table,b)]*.

on(b,c) As can easily be verified, starting with this subgoal results in the even worse plan *[move3(b,table,c), move2(b,c,table), move2(c,a,table), move3(a,table,b), move2(a,b, table), move3(b,table,c), move3(a,table,b)]*.

Obviously, the generated plans are pretty awkward and the shortest plan is $\{$*move2(c, a,table), move3(b,table,c), move3(a,table,b)*$\}$. Hence, we can conclude that the GPS procedure is incomplete, if we define completeness such that all minimal plans are found. The question we pursue now is, how completeness can be assured.

First, we observe that the goal $\{$*on(a,b), on(b,c)*$\}$ is not a complete description of a consistent world because it does not contain the fact *on(c,table)*. Second, we observe that if the goal would be complete in this sense, i.e., would comprise the facts *on(a,b)*, *on(b,c)*, and *on(c,table)*, then the GPS procedure would easily find the optimal plan when starting with the goal *on(c,table)*. Hence, we might conjecture that GPS results in a complete planner, if the goal describes a consistent world. Since the user of the planner should not be concerned with this restriction, we use the repair mechanism of the previous section to compute the complete goal world(s).

[1]The violation of consistency constraints is not treated by STRIPS.

For a given goal G, we get a complete description of the goal world W_G, by adding G to W_0. This results in

$$W_G = \{on(a, table), on(c, a), on(b, table), on(a, b), on(b, c)\}.$$

Obviously, several constraints are violated: block a is at two places, thus '$on(a, table)$' must be deleted. For the same reason '$on(b, table)$' must go. We now have W_G = { *on(a,b), on(b,c), on(c,a)*}, which is an impossible 'circular' tower, therefore '$on(c, a)$' also disappears. The result is—in this case—a world holding only the goal. However, this still violates a constraint, namely that every object must be at a certain place. Block 'c' has no place, so we must find one for it. The only choice here is the addition of '$on(c, table)$', so we finally get the goal state $W_G = \{on(a, b), on(b, c), on(c, table)\}$ of figure 2. As it can be seen, we arrived at a complete and consistent description of the goal state.

To summarize, the key idea of our approach is to borrow ideas from deductive database technology for modeling worlds in planning and to use techniques for detecting and resolving inconsistencies. By these means, we compute a complete description of a possible goal world and use this information to guide a dedicated (linear) planning algorithm. As the main driving force is regaining consistency, we call our approach **Consistency Driven Planning (CDP)**.

The next subsection gives the definition of the basic CDP procedure which is a generalization of GPS including the repair mechanism. This basic procedure is rather inefficient since too many search nodes are generated during a planning process. To overcome this deficiency, some pruning techniques are incorporated. Then, completeness results are stated. A performance evaluation of the different pruning algorithms will conclude the subsection.

3.2 The Basic CDP Procedure

3.2.1 Basic CDP-Algorithm

The main difference between our approach and most planners which are based on a STRIPS–like formalism ([5]) is that the CDP planner is able to handle formulas which occur either as preconditions or as consistency constraints of a domain. The repair system allows to map general formulas to a set of ground literals. This capacity is required during the elimination of critical operators of a plan by insertion of new ones. If *op* is a critical operator with respect to a world W the following set of repairs will be generated:

$$\begin{aligned} repairs_i(W, op) \;:=\; \{R_1 \cup R_2 \mid\; & R_1 \in repairs_i(W, pre(op)) \\ \&\; & R_2 \in repairs_i(op(R_1(W)), W^c) \\ \&\; & R_2 \cup post(op) \text{ free of contradictions} \\ \&\; & R_1 \cup R_2 \text{ free of contradictions}\} \end{aligned}$$

The index i denotes the **repair level**. The function $repairs_i$ is based on CAUSES_i which was defined in subsection 2.4:

$$repairs_i(W, \Delta) \;:=\; \Gamma^{-1}(\text{CAUSES}_i((W^a, W^r, \Delta))$$

The following algorithm shows how this mechanism can be incorporated into a simple planner:

Algorithm 3.3 (basic planning algorithm)

(1) *procedure Basic-CDP(W_0,G)*
(2) *init* plan := [*goal*] *where* pre(*goal*) := G *and post*(*goal*) := $\emptyset$
(3) *while* plan *is no solution do*
(4) *determine a critical* op_l *of* plan = $[op_1, \ldots, op_n]$: $l := SC(\text{plan})$
(5) *let* $\forall j = 1 \ldots n : W_{j+1} = \mathrm{op}_j(W_j)$
(6) Rep := $\text{repairs}_i(W_{l-1}, \mathrm{op}_l)$
(7) *choose a subgoal* $g \in \bigcup_{rep \in Rep} \text{rep}$
(8) *choose an operator* op: $g \in \text{post}(\text{op})$
(9) *choose a position* j: $j \leq l$
(10) plan := $\text{plan}[1 \ldots j-1] \circ [\text{op}] \circ \text{plan}[j \ldots n]$;
(11) *return* plan

The lines 7, 8 and 9 are choice points, i.e., the planner has to choose one out of several possible results and has to store the remaining results in order to allow backtracking. In line 4, the selection of a critical operator is deterministic. Which critical operator is selected depends on the applied **selection function** SC. The special operator *goal* allows a uniform treatment of the goal G. If *goal* is selected as a critical operator the applications the repairs computed in line 6 result in completions of the goal, i.e., we perform a goal completion. Since the completion of the goal is done in respect to the last world W_{n-1} of the plan[2], the completed worlds always reflect the actual situation of the planning process.

In the following example we demonstrate how the basic planning algorithm generates a solution for the Sussman Anomaly. In order to keep the example short we assume that the algorithm always performs optimal choices.

Initialization

The planning algorithm starts with the plan [*goal*]. The precondition of the special operator *goal* is the goal $G = \{on(a,b), on(b,c)\}$.

First execution of the while body

Step 1.1: (choose a subgoal) *To satisfy the precondition of the operator goal a list of repairs is computed by the repair mechanism. After that, the planner chooses a subgoal at the first choice point.*

The repair mechanism finds exactly one repair

$$\begin{aligned} R \;=\; & \{add(on(a,b)), add(on(b,c)), add(on(c,table)), \\ & del(on(c,a)), del(on(a,table)), del(on(b,table))\} \end{aligned}$$

for the world W_0 and the goal G such that $C(R(W_0)) \cup W_0^c \cup G$ is consistent. Note, that the repair mechanism derives a new fact $on(c,table)$ from W_0 and G. The fact $on(a,b)$ is chosen as a the first subgoal g.

[2]The goal operator does not change the world W_{n-1}.

Step 1.2: (choose an operator) *The task for the planner is to find an operator 'performing' the subgoal g choosen by the previous choice point. If there are several possible operators a single one is selected.*

The operators $move(a, c, b)$ and $move(a, table, b)$ can be used to achieve $g = on(a, b)$. The second operator is chosen.

Step 1.3: (choose a position) *The algorithm inserts the selected operator in the generated plan at some position before the critical operator.*

The only possible position for $move(a, table, b)$ is directly before *goal*.

Second execution of the while body

The selection function SC determines $move(a, table, b)$ as critical operator.

Step 2.1: (Determine repairs) The subgoal $g = \neg on(c, a)$ is chosen from the only repair $\{\neg on(c, a), on(c, table)\}$.

Step 2.2: (choose an operator) Analogously to step 1.2, the operator $move(c, a, table)$ is chosen.

Step 2.3: (choose a position) Again, the introduction of the chosen operator into the plan is deterministic. $move(c, a, table)$ is inserted in the first position of the generated plan:

$$[move(c, a, table), move(a, table, b), goal()]$$

Third execution of the while body

goal() is the only critical operator.

Step 3.1: (choose a subgoal) The precondition of *goal* and the consistency constraints of the domain hold after the execution of the repair $\{on(b, c), \neg on(b, table)\}$. The planner choses $on(b, c)$.

Step 3.2: (choose an operator) For the subgoal $on(b, c)$ the second choice point selects $move(b, table, c)$.

Step 3.3: (choose a position) For $move(b, table, c)$ exist three possible positions. By insertion of $move(b, table, c)$ in the second position of the plan we generate a solution

$$[move(c, a, table), move(b, table, c), move(a, table, b), goal()]$$

for the planning problem, since the intermediate worlds are consistent and the preconditions of all operators are satisfied.

3.3 Enhancements of the Basic Planner

Three different planning strategies, which exploit the fact that the planner allows a deterministic selection of a critical operator of a plan, have been implemented. The 'leftmost critical operator' (LMC) strategy always selects the critical operator op_i with the smallest index i; the 'rightmost critical operator' (RMC) strategy uses the critical operator with the highest index. The third strategy GPS is borrowed from the General Problem Solver ([3]). GPS is a LMC strategy where a new operator is always inserted directly before the selected critical operator, i. e., GPS replaces the nondeterministic choice point at line 9 by the deterministic statement $j := l$.

On one hand, it is well-known that if the search space of an existing planning strategy is pruned, no minimal plans or — even worse — no plans at all may be found. On the other hand, the search space often has to be pruned in order to get the necessary performance to solve even simple problems. In the following we introduce three powerful pruning heuristics which preserve completeness for LMC and RMC.

The first pruning technique **hitting-set (HS)** restricts the number of alternatives at the first choice point of the algorithm above by using a minimal hitting set of all computed repairs. Note, that the set of repairs can be large.

Example 3.4 *If we extended the initial world of the conjunctive goal problem by the facts* $on(d_1, \text{table}), \ldots, on(d_n, \text{table})$ *we would get* n+1 *repairs:*

$$\{\{on(a,b), on(b,c), on(c,x), \neg on(c,a), \neg on(a,\text{table}), \neg on(b,\text{table})\} \mid x \in \{\text{table}, d_1, \ldots, d_n\}, i = 1 \ldots n\}$$

consisting of $6+n$ *different literals, whereas all minimal hitting-sets, e.g.,* $\{on(a,b)\}$*, only consist of one literal.*

The branching factor of the first choice point is decreased by restricting the subgoals to a minimal hitting set $HS(Rep)$ of all computed repairs. This idea is incorporated into the planner above by replacing line 7 by

(7.a) determine a minimal hitting set of $Rep = \{R_1, \ldots, R_k\}$: hs:=HS($Rep$);
(7.b) **choose** a subgoal $g \in hs$

The selection of a hitting set out of several possible ones in line 7.a is done deterministicly therefore no backtracking capabilities are necessary for this step.

The second pruning technique reduces the branching factor at the second choice point by rejecting operators that destroy an already achieved subgoal. As an example consider the following incomplete plan for the conjunctive goal problem:

$$[move(c, a, table), move(a, table, b), move(b, table, c)].$$

The first two operators successfully achieve the subgoals $on(c, table)$ and $on(a, b)$. The third operator tries to achieve $on(b, c)$, but is not applicable since the precondition of $move(b, table, c)$ requires that the block b is clear. The only way to achieve $\forall x_4 : \neg on(x_4, b)$ is to destroy $on(a, b)$ by inserting additional operators. This plan does not lead to an optimal solution therefore its expansion should be averted.

In order to prevent this clobbering of an already achieved subgoal we use the well-known technique of **protected subgoals**[3] **(PSG)** which only allows the insertion of operators into positions where they do not violate previously defined producer–consumer relations. Additionally, the insertion of an operator at a position j is prevented if between j and the critical operator already lies a potential destroyer. The GPS-planner is extended by an additional variable *psg* which lists the currently protected producer–consumer relations. The initialization of Basic-CDP is extended by $psg := \emptyset$ and line 9 in the planner is replaced by:

(9.a) **choose** a position j for *op* such that:

- $j \leq l$
- the insertion of *op* at position j does not violate any in *psg* specified producer–consumer relations
- between j and l are no potential clobbers of g in *plan*

(9.b) $psg := psg \cup \{$"*op* produces g for op_i"[4]$\}$;

The expansion of the above plan can only partially be prevented by this modification since the second operator $move(a, table, b)$ could be inserted in order to achieve either $on(a, b)$ or $\neg on(a, table)$. In the second case, it would be possible to remove a from b again. If we protected several subgoals for one producer, we could also prevent this case. But so far the problem of efficiently deciding whether an operator can achieve more than one subgoal and therefore the problem of protection of more than one subgoal for an operator is unsolved. In the blocks world this is even the main problem. In [7] Gupta and Nau show that the worst case complexity of the computation of an optimal solution is determined by the problem of deciding whether a block has to be moved by one or two *move*-operators to its goal position.

The third pruning technique concerns the computation of repairs for a critical operator op_l. The selection of a subgoal g can be interpreted as a preference of those repairs R_k of *Rep* which contain g. If during further computations op_l is again selected, we should respect the former decision and try to choose an unachieved subgoal of the 'preferred' repairs R_k which contain g. This consideration is the foundation of the **postponed repair computation (PRC)** pruning technique. Each operator *op* of a plan has a set of associated repairs *assoc-rep*(op). Initially, this set is empty. If an operator is selected by SC, the associated set of repairs is initially used to determine a subgoal. Only if all associated repairs are already achieved or if no repair is associated a new set of repairs is computed. After the selection of a subgoal, those repairs which contain the selected subgoal are linked to the critical operator. We integrate this technique into the planner through the replacement of line 6 by

(6.a) $Rep := assoc\text{-}rep(op_l)$
(6.b) **if** $\exists R \in Rep : W_{l-1} \models R$
(6.c) **then** $Rep := repairs_i(W_{l-1}, op_l)$

[3] see, e.g., [14]

[4] we assume that operators which occur several times in a plan can be distinguished from each other

and through the insertion of a 11th and 12th line:

(11) $assoc\text{-}rep(op_l) := \{R \in Rep \mid g \in R\}$
(12) $assoc\text{-}rep(op) := \emptyset$

The goal operator *goal* is initialized in line 2 by a new statement $assoc\text{-}rep(goal) := \emptyset$. Since associated repairs may no longer comply with the definition of the term 'repair' (e.g., they may not be minimal) before the application of HS, the literals already achieved have to be removed from the associated repairs.
Figure 4 shows the entire planner with all three pruning techniques incorporated.

(1) **enhanced CDP–Planner** Enhanced-CDP(W_0,G)
(2) **init** $plan := [goal]$ where $pre(goal) := G$ and $post(goal) := \emptyset$
(3) $psg := \emptyset$
(4) $assoc\text{-}rep(goal) := \emptyset$
(5) **while** *plan* is no solution **do**
(6) determine a critical op_l of $plan = [op_1, \ldots, op_n]$: $l := SC(plan)$
(7) let $\forall j = 1 \ldots n : W_{j+1} = op_j(W_j)$
(8) $Rep := assoc\text{-}rep(op_l)$
(9) **if** $\exists R \in Rep : W_{l-1} \models R$
(10) **then** $Rep := repairs_i(W_{l-1}, op_l)$;
(11) determine a minimal hitting set of $Rep = \{R_1, \ldots, R_k\}$:
$hs := HS(\{\ \{l \in R_i | W_{l-1} \not\models l\}\ \mid\ R \in Rep\})$;
(13) **choose** a subgoal $g \in hs$;
(14) **choose** an operator op: $g \in post(op)$;
(15) **choose** a position j for op such that:
- $j \leq l$
- the insertion of op at position j does not violate any in psg specified producer–consumer relations
- between j and l are no potential clobbers of g in *plan*

(20) $psg := psg \cup \{$"op producess g for op_i"$\}$;
(21) $plan := plan[1 \ldots j-1] \circ [op] \circ plan[j \ldots n]$;
(22) $assoc\text{-}rep(op_l) := \{R \in Rep \mid g \in R\}$
(23) $assoc\text{-}rep(op) := \emptyset$
(24) **return** *plan*

Figure 4: The entire CDP-planning algorithm

We now state some completeness results on the different variants of the CDP procedure. For that, we first define the notion of completeness. The proofs of following completeness theorems can be found in [10].

Definition 3.5 (Completeness) *A planning strategy is* **complete** *if for every minimal solution of a solvable planning problem there exists a sequence of choices for the choice points such that the planning strategy finds this solution.*

Theorem 3.6 (Completeness of the Basic CDP Procedure) *The basic CDP procedure is complete for every repair level i.*

Theorem 3.7 (Completeness of Combination) *The basic planner together with with every combination of the strategies HS, PSG, and PRC is complete.*

3.4 Empirical Evaluation of the Planner

In the last section, we have presented different planning strategies and pruning techniques. In order to evaluate the usefulness of these extensions to the basic planner, we have conducted a series of experiments with several planning domains. In this section, we present some of these results.

The tables 1 and 2 summarize performance results for the conjunctive goal problem, for the river crossing example, and for the railway example. In the river crossing example, a farmer has to move a goose, a dog, and bag of corn from the left side of a river to the right side. The farmer has a small boat which can carry himself and at most one further object. Neither the dog and the goose, nor the goose and the corn may stay on the same side without the farmer, since the dog would eat the goose and the goose the corn.

In the railway example an engine has to transport a waggon from one end of a small marshalling yard to the other end. Four operators describe the different actions. The operator *move* moves the engine whereas *transport* moves the engine and the wagon. The last two operators describe the coupling-up and decoupling of wagon and engine.

All tested configurations incorporate goal completion and pure breadth first search. No heuristic information is used in the search. Every choice at any choice point induces the generation of all successors of the corresponding node of the search tree. Note, that we count the generated nodes and not the expanded (or *closed*) nodes.

		GPS	LMC	GPS_{hs}	LMC_{hs}	$GPS_{hs,psg}$	$LMC_{hs,psg}$
conj. goal	nodes	306	287	36	30	36	30
	run time	7.5	6.4	2.8	1.9	2.7	1.9
river cross.	nodes	>2001	>2001	262	535	208	331
	run time	54	56	11	22	6.9	11
rail-way	nodes	>2001	>2001	76	326	76	326
	run time	89	92	5.5	17	5.6	17

Table 1: Number of generated search nodes and run time for different strategies. All runs include the PRC-strategy.

The application of the hitting set strategy results in the greatest reduction of the search space. If we look at table 2 which shows the branching factors for the three choice points, we can see why the minimal hitting set strategy leads to such an improvement. Its application reduces the branching factor of the subgoal choice point on average by a factor of 3. Table 2 also shows that the completeness of the LMC planning strategy does not increase the cost. In comparison with the GPS search strategy the LMC strategy has a branching factor of the choice point 'choose a position' which is only about 1.2 times higher than that of the (incomplete) GPS strategy.

	choice point	GPS	LMC	GPS_{hs}	LMC_{hs}	$GPS_{hs,psg}$	$LMC_{hs,psg}$
conj. goal	subgoal	2.9	3.2	1	1	1	1
	operator	2	2	2	2	2	2
	ins. pos.	1	1.5	1	1.4	1	1.4
river cross.	subgoal	2.4	2.7	1.3	1.2	1.1	1.2
	operator	1.3	1.4	1.6	1.7	1.9	1.7
	ins. pos.	1	1.2	1	1.3	1	1.3
rail-way	subgoal	5.9	5.9	1	1	1	1
	operator	3.9	3.9	3.7	5.4	3.7	5.4
	ins. pos.	1	1.1	1	1.2	1	1.3

Table 2: Branching factors of the three choice points for different strategies. For all runs the PRC-strategy is switched on.

If additional domain independent heuristics[5] and a best-first search are applied, the number of generated nodes for the conjunctive goal problem can be reduced to 20 nodes, 52 nodes for the river crossing problem, and 39 nodes for the railway example.

3.5 Summary

We have specified different planning procedures together with a set of pruning heuristics which still allow complete planning procedures. The approach was heterogeneous in nature since the planning procedure was a dedicated one which relied on consistency maintenance. The hitting set strategy was revealed to be the most effective pruning technique.

We now come to the disadvantages of the approach. Consider a blocks world with many blocks. When a block has to be removed from another block by the operator *move*, the actual place where to put the block is open. Every other block and the table are possible places. The planner made this decision immediately and generates alternatives depending on the place where the block is actually put. This immediate instantiation of operators leads to too many alternatives. A better way is to defer the decision until more information is available. Therefor, variables have to occur within partial plans.

A second disadvantage is that positions of operators are also chosen immediately. This can lead to too many alternatives if two subgoals are independent. Hence, it might be promising not only to consider linear plans but also non-linear plans, i.e. those, which contain operators which are not yet ordered.

The goal of the next section is to design a planner which

- is homogeneous,
- consists of only a single choice point guided by the hitting set technique,
- defers variable instantiation, and
- allows non-linear plans.

[5]the heuristics include the preference of operators which achieve several subgoal or the preference of uncritical positions for the insertion of operators

4 CDP: The Homogeneous Approach

The previous planner consists of two levels:

1. the dedicated planning algorithm which worked on top of
2. a deductive database with consistency maintenance capabilities.

A sequence of operators is a correct plan iff the precondition of each operator is fulfilled, the intermediate worlds are consistent, and the goal world is reached by the sequence. Here, consistency and preconditions are described by logical formulas. The correctness of the plan is built-in into a specialized planning algorithm.

In [4] an event calculus is used to describe the changes of operators resulting from their application to worlds. The correctness of a plan can then be *deduced.* A planning procedure can be derived by formalizing/modeling the notion of correct plan by facts, rules, and constraints. A plan can then be derived by repairing the violations of the constraints.

This section follows this approach. More specifically, the first subsection models the planning domain whereas the second shows how plans are actually derived.

4.1 Modeling Planning

In order to model planning, we need the following base predicates:

before models the fact that a certain operator occurs before some other operator. We will not only allow linear but also non-linear plans. More specifically, a plan will be modeled by a set of positions. Each position has a set of assigned operators.

active Among the operators assigned to a position the operator which actually occurs in the considered/resulting plan is modeled through the predicate *active.* Since operators are not bound to certain positions by themselves, *active* is a binary predicate taking the operator and the position as arguments.

value In the CDP planner, as introduced in the last section, all operators have been fully instantiated. This can lead to major inefficiencies. Hence, within the homogeneous approach we model variable assignment to operator arguments explicitly. The predicate *value* is binary with variable-modeling constants and object constants as arguments.

equal expresses that two variables must have an equal assignment.

Let us give a small example how a plan is described. In Figure 5 a simple plan within the blocks world is modeled. The special operators *start* and *goal* are used to simplify the formalization:

start introduces with its postcondition *post(start)* the initial world;

goal introduces with its precondition *pre(goal)* the planning goal $\{on(c, b)\}$.

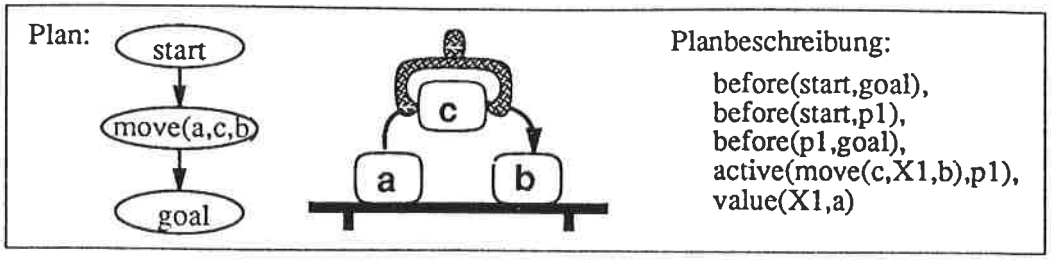

Figure 5: An example for a plan description

The next step in modeling the planning domain is to express correctness of plans within the formalization. In [1] a truth criterion is introduced which allows to check the validity of an assertion w.r.t. a non–linear plan without exploiting all linearization of the plan. This criterion is modeled with a set of rules. We express that a literal l holds immediately before the execution of an operator of a position p by *holds(l,p)*. Verbally, the truth criterion expresses the following:

> $holds(l, p)$ is valid, if there exist a producer op of a position p' with $l \in post(op)$ within the considered Plan P and there exists no destroyer op' between p and p' such that $\bar{l} \in post(op')$.

This can be captured as follows:

- (EST-LIT)

  $holds(\text{base literal } l, p) \Leftarrow before^*(p', p),\ \ effect(p', l),\ \ \neg destroyer(p', p, l)$

- (DESTR)

  $destroyer(e, u, l) \Leftarrow \neg before^*(c, e),\ \neg before^*(u, c),\ effect(c, l^c)$

where *before** is the transitive closure of *before* and *effect* models the postconditions of active operators (see below). The predicate *destroyer* models destroyer operators as used in the description of the truth criterion. Occurring variables are typed, e.g., the variables p and p' have the domain Pos_Σ and l has the domain Lit_Σ. The domain could also be determined by *restriction literals* ([12]).

Consistency constraints are checked after the execution of an operator. In linear plans this can be achieved by simply checking the resulting world. For non-linear plans this is more complicated since the resulting worlds are not necessarily unique. Hence, a special truth criterion must be employed. Analogously to *holds*, we define a predicate *holds-after*:

- (EST-AFTER-LIT1)

  $holds\text{-}after(\text{base literal } l, p) \Leftarrow effect(p, l)$

- (EST-AFTER-LIT2)

  $holds\text{-}after(\text{base literal } l, p)$
  $\Leftarrow before(p', p),\ effect(p', l),\ \neg destroyer(p', p, l),\ \neg effect(p, l^c)$

The *effect* of an operator depends on the current assignment for variables and the current equations as modeled by *value* and *equal*. The following three rules capture this dependency:

- (EFFECT1) $effect(p, l) \Leftarrow active(op, p)$ where $l \in post(op)$ and op associated to p.
- (EFFECT2) $effect(p, l[c \leftarrow V]) \Leftarrow value(V, c),\ effect(p, l)$
- (EFFECT3) $effect(p, l[V \leftarrow V']) \Leftarrow equal(V, V'),\ effect(p, l)$

The predicate *holds* also has to capture whole formulas (not only literals). For that, the derivation process has to be built into the rule set such that the special semantic $\models_{worlds}$ of worlds, which includes the unique-name, domain-closure, and closed-world assumptions, is captured. The following theorem states when a rule set is correct and complete:

Definition 4.1 (correctness and completeness of R_{derive})
Let $\langle W^a, W^r, W^c \rangle$ be a world over the signature $\Sigma = (K_\Sigma, Pr_\Sigma, \alpha_\Sigma, Pos_\Sigma)$, R_{derive} a rule set defining the predicate holds *and φ a closed formula over Σ.*

R_{derive} is a correct and complete axiomatization of the the semantic of worlds, if and only if:

$$\text{comp}(\{\text{holds}(l, p) \mid l \in W^a \cup \overline{(Fak_\Sigma^{base} \setminus W^a)}\} \cup R_{derive}) \models \text{holds}(\varphi, p) \ \ \textit{iff} \ \ W \models_{worlds} \varphi$$

An example for a correct and complete derivation scheme has been given in Section 2. A direct definition of derivation trees in a rule set R_{derive} is:

(ALL)	$holds(\forall x : \varphi, p) \Leftarrow \bigwedge_{c \in K_\Sigma} holds(\varphi[c \leftarrow x], p)$
(EX)	$holds(\exists x : \varphi, p) \Leftarrow holds(\varphi[c \leftarrow x], p)$
	where $c \in K_\Sigma$ constant of object domain
(RULE)	$holds(a, p) \Leftarrow holds(\bigwedge a_i, p)$
	where $a \Leftarrow a_1 \ldots a_n$ fully instantiated object rule of W^r
(NOT)	$holds(\neg a, p) \Leftarrow \bigwedge_{a \Leftarrow a_1 \ldots a_n \text{fully instantiated rule of } W^r} holds(\neg a_1 \vee \ldots \vee \neg a_n, p)$
(AND)	$holds(\varphi \wedge \psi, p) \Leftarrow holds(\varphi, p), holds(\psi, p)$
(OR)	$holds(\varphi \vee \psi, p) \Leftarrow holds(\varphi, p)$
	$holds(\varphi \vee \psi, p) \Leftarrow holds(\psi, p)$

Note that the unique-name and domain-closure assumptions are contained within the axioms. The closed-world assumption is gained by the completion of R_{derive}. The axioms for *holds-after* can be derived by replacing *holds* by *holds-after* within the above rule set. Subsequently, we concentrate on *holds* since all results can be derived analogously for *holds-after*.

With the above rule sets the rule interpreter can decide whether a certain precondition φ holds before the execution of an operator op at position p by checking $holds(\varphi, p)$. The *uncritical* application of an operator can now be formalized by the rule:

- (NC) $not\text{-}critical(op, p) \Leftarrow holds(\bigwedge_{\varphi \in pre(op)} \varphi, p),\ holds\text{-}after(\bigwedge_{\varphi \in W^c} \varphi, p)$

We are now ready to axiomatize the correctness of plans. First, we state the condition that every occurrence of an active operator must be uncritical:

- (PC) $\forall\, p, op : active(op, p) \ \Rightarrow \ not\text{-}critical(op, p)$

The other constraints are general planning conditions which must be fulfilled for every plan:

- There is no cycle within the positions (cycle condition):

  (CYC) : $\forall\, p : position(p) \Rightarrow \neg before^*(p,p)$

- All positions lie between the start and the goal position:

  (S-G) : $\forall\, p : p \neq p_{start},\ p \neq p_{goal} \Rightarrow before^*(p_{start},p) \wedge before^*(p,p_{goal})$

- Variable assignments are unique (unique variable value):

  (UVAR): $\forall\, v,c,c' : value(v,c),\ value(v,c') \Rightarrow c = c'$

- Equal variables have the same assignment (equal variable value):

  (EVV): $\forall v,v',c,c' : equal^*(v,v'),\ value(v,c),\ value(v',c') \Rightarrow c = c'$

  where $equal^*$ is the transitive closure of *equal*.

- There is at most one active operator at each position:

  (ACT): $\forall\, p, op, op' :\ active(op,p),\ active(op',p) \Rightarrow\ op = op'$

- The start und the goal operator are always active:

  (S&G-ACT): $active(start, p_{start}) \wedge active(goal, p_{goal})$

The rules and constraints introduced so far constitute an axiomatization of the domain "planning". Analogously to the description of a world as a triple of facts, rules, and constraints, a plan is described through a triple $P = \langle P^a, P^r, P^c \rangle$. In order to distinguish between the formalization of worlds of a domain and the domain 'planning', we denote the former by **object domain** and the latter by **meta domain**. Coupling these domains is realized by the application of the

- object rules within the rules (RULE) and (NOT) and
- the object constraints and operator preconditions within the rule (NC).

A planning problem is described by a start operator *start* whose postcondition contains the initial world, and a goal operator *goal* whose precondition reflects the planning goal:

$pre(start) = \emptyset, post(start) = \{l \mid l \in W^a \cup \overline{(Fak_\Sigma^{base} \setminus W^a)}\}$
$pre(goal) = G, post(goal) = \emptyset$

We are now able to state an important result which allows us to determine the usefulness of our plan axiomatization. Verbally, the consistency of a plan P is a sufficient criterion for the correctness of each linearization of P:

Theorem 4.2 (correctness of the planning formalization)
Let $P = \langle P^a, P^r, P^c \rangle$ *be a plan for a planning problem* $\langle W_0, OP, G \rangle$, *and* P *(GPC)-consistent,* $\sigma = \{X/c \mid \text{value}(X,c) \in P^a \text{ or } \text{equal}^*(X,X'), \text{value}(X',c) \in P^a\}$ *and* $\{(\text{op}_0, p_0), \ldots, (\text{op}_n, p_n)\} = \{(\text{op}\sigma, p) \mid \text{active}(\text{op}, p) \in P^a\}$
If P *is (PC)-consistent, then every linearization of the plan described by* P^a *is correct, i.e., for all linear plans* $[\text{op}_{i_0}, \ldots, \text{op}_{i_n}]$ *such that* $\text{op}_0 = \text{start}$, $\text{op}_{i_n} = \text{goal}$ *and* $p_{i_1}, \ldots, p_{i_n}$ *is a linarization of* before* *the following holds:*

$$W_{k+1} := \text{op}_{i_k}(W_k) \text{ consistent and } \text{pre}(\text{op}_{i_k}) \text{ is valid in } W_k$$

We omit the proof.

The reverse implication of above theorem is not true. Figure 4.2 contains a counter example, where every linearization of the plan is correct but the plan is not (PC)-consistent. The non–linear ordering of the operators op_1 and op_2 causes a negative interaction w.r.t. the two meta facts $holds(a, p_{goal})$ and $holds(b, p_{goal})$.

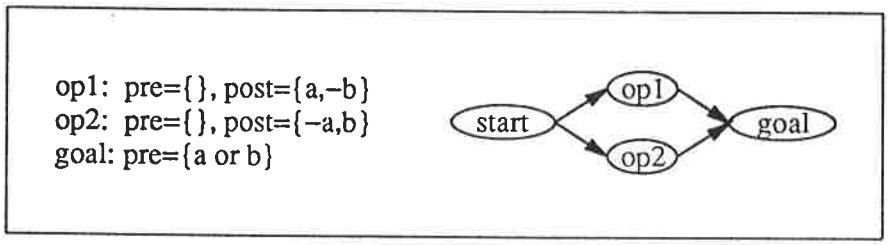

Figure 6: Example of a plan of which all linearizations are correct, but which is not (PC)–consistent

Nevertheless, for practical reasons the *weak completeness* of our plan axiomatization is sufficient:

Theorem 4.3 (weak completeness of the planning formalization)
For every correct plan $[\text{op}_{i_0}, \ldots, \text{op}_{i_n}]$ *for a planning problem* $\mathcal{P} = \langle W_0, OP, G \rangle$, *there exists a consistent plan* $P = \langle P^a, P^r, P^c \rangle$ *for* $\mathcal{P}$.

We omit this proof, too.

4.2 Plan Generation

Within this subsection we want to demonstrate, that the above formalization is not only useful in specifying the correctness of plans and its verification, but also to actually derive correct plans. The central idea is to apply the consistency maintenance procedure of Section 2.

Again, the main tools are derivation trees. More specifically, derivation trees for the correctness condition (PC). Figure 7 contains a partial derivation tree for the plan of Figure 5. This partial tree is used for proving the precondition *on(c,b)* of the goal operator *goal*. Within the nodes, the instantiated meta rules of P^r are denoted by their names.

The task of the planner is to generate a consistent solution plan for a given planning problem $\langle W_0, OP, G \rangle$ where a plan P^a is consistent if (GPC) and (PC) are satisfied.

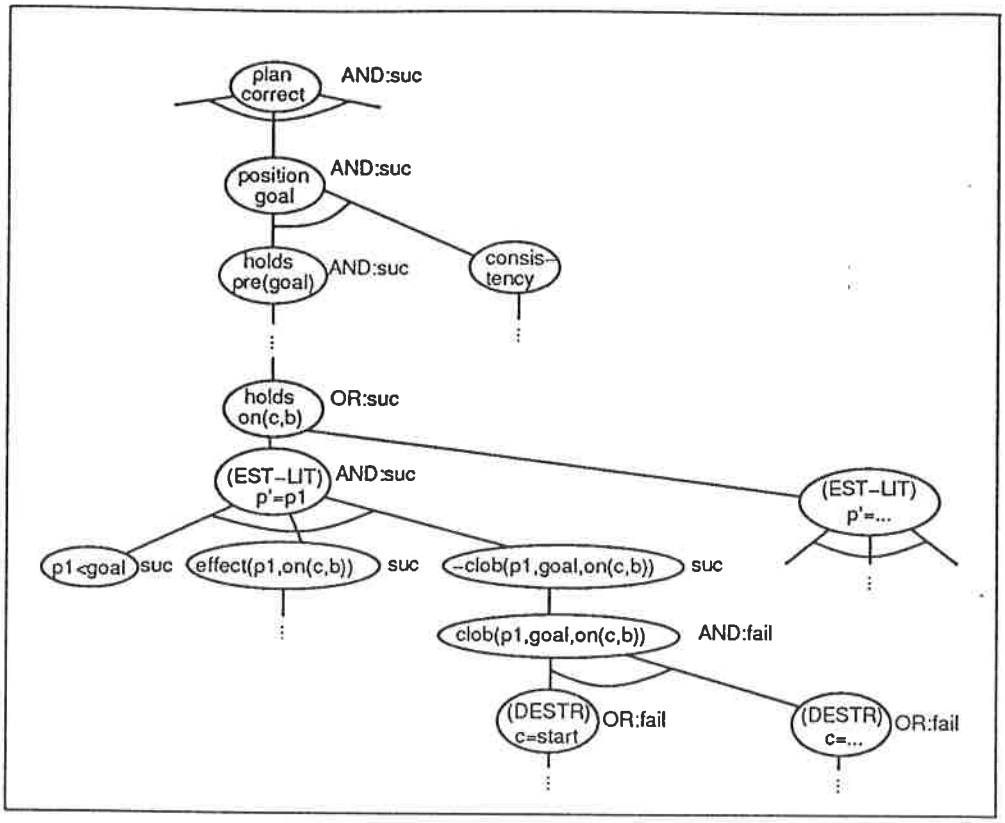

Figure 7: A part of the derivation tree for the example plan

Whereas (GPC) is easy to verify and guarantee, the main challenge is (PC). There exist two different causes due to which (PC) can be violated:

1. There exists no solution tree whose leafs are contained in $P^a \cup \{\neg a \mid a \text{ base atom}, a \notin P^a\}$. In this case, the simple solution to the problem is to modify P^a accordingly.

2. Positions with appropriate active operators are missing.

The latter condition is the real challenge. Missing operators, or those which cannot be activated because of the constraint (ACT), lead to the problem that there exists no producer or only producers followed by destroyers for an object base literal l. To account for this situation, a new position p_{new} has to be introduced. A set of partially instantiated operators containing l in their postcondition can be assigned to p_{new} in a fixed way. Then, these operators are those which can potentially be activated (see EFFECT1). Introducing a new position with new operators allows to generate new instances of (NC), (EST-LIT), (EST-AFTER-LIT1,2) and (DESTR), and, hence, leads to an expansion of the derivation tree. This expanded derivation tree then, possibly, contains the potential of gaining consistency by a simple modification of P^a. Note that the introduction of new positions occurs, in general, already at the start of the planning process where only the two positions p_{start} and p_{goal} for the operators *start* and *goal* exist.

The reason for introducing new positions is rule (EST-LIT). Its completion results in

$$\mathit{holds}(\text{base literal } l, p) \Leftrightarrow \exists p' \in Pos_\Sigma : (\mathit{before}^*(p', p),\ \mathit{effect}(p', l),\ \neg \mathit{destroyer}(p', p, l))$$

Replacing $destroyer(p', p, l)$ by the body of (DESTR) yields

$$holds(\text{base literal } l, p) \Leftrightarrow \exists p' \in Pos_\Sigma : (\; before^*(p', p), \; effect(p', l), \; \forall p'' \in Pos_\Sigma : (before^*(p'', p') \vee before^*(p, p'') \vee \neg effect(p'', l^c)))$$

Obviously, only in (EST-LIT) occur position variables which are actually quantified by an existential quantifier. This requires that the planner must be able to introduce new positions during the planning process. The derivation tree should also be used to determine new positions. Therefore, a new successor with the label *new–position* is assigned to each node with a label of the form $holds(l, p)$ and where l is an object base literal (see Figure 8).

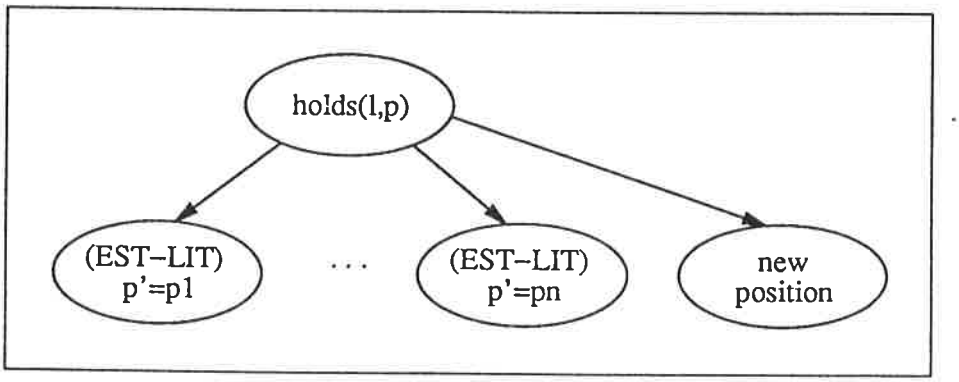

Figure 8: Extension of the derivation tree by a new leaf "*new–position*"

This enhanced derivation tree is denoted by $DT_\Sigma^{pos}(P)$. A node "*new–position*" is a leaf whose label cannot directly be transformed into a meta base fact, which could be added to P^a. If during the planning process a *new–position* leaf is to be satisfied in order to achieve consistency (validity of (PC)), Pos_Σ is extended by a new position p_{new} and the selected *new–position* node n is eliminated from the derivation tree. The predecessor n' of n contains a label $holds(l, p)$. According to this literal l, all operators with l in their postcondition are now associated to p_{new}. Finally, the derivation tree $DT_\Sigma^{pos}(P)$ is adjusted for the new signature $\Sigma := \Sigma \cup \{p_{new}\}$. The resulting planning procedure can be described as follows:

Algorithm 4.4 (basic planning algorithm)

(1) *init* pre(start) $:= \emptyset$, post(start) $:= \{l \mid l \in W^a \cup \overline{(Fak_\Sigma^{base} \setminus W^a)}\}$
(2) pre(goal) $:= G$, post(goal) $:= \emptyset$
(3) $Pos_\Sigma := \{p_{start}, p_{goal}\}$
(4) $P^a :=$ {active(start,p_{start}), active(goal,p_{goal}), before(p_{start},p_{goal})}
(5) *compute initial derivation tree* $DT_\Sigma^{pos}(P)$
(6) ***while*** *P not consistent* ***do***
(7) ***choose*** *leaf n of* $DT_\Sigma^{pos}(P)$ *where meta literal l is not satisfied by P*
(8) ***if*** *n is a* new–position *node*

(9) *then create a new position* $p_{new} \notin Pos_\Sigma$
(10) *associate operators*
$\{\text{op} \mid l \in \text{post(op)}$ & op *min. instanciated*$\}$ *to* p_{new}
(11) $Pos_\Sigma := Pos_\Sigma \cup \{p_{new}\}$,
(12) *adopt* $DT_\Sigma^{pos}(P)$ *to the extended signature*
(13) **else** $P^a := P^a \cup \{l\}$

First note that this algorithm is still nondeterministic. Nevertheless, opposed to the previous planners, it contains only a single choice point. Further, this choice point can be driven by the hitting set heuristic. In order to do so, line 7 is replaced by

- Compute a set of repairs $r_1, \ldots, r_n$ for the derivation tree $DT_\Sigma^{pos}(P)$ where the ground meta base literals $P^a \cup \{\neg a \mid a \text{ base atom}, a \notin P^a\}$ describes the valid leaves
- choose a node from the hitting set of $r_1, \ldots, r_n$

Also, the other goals

1. introduction of variables to avoid immediate binding
2. non-linear plans to avoid immediate ordering

are fulfilled by this algorithm.

5 Conclusion

After shortly reviewing consistency maintenance, its usefulness in planning has been motivated, and several planning procedures exploiting consistency maintenance have been given. Theoretical as well as empirical results confirmed the track. Nevertheless, some possible problems of these planning procedures where pointed out. In order to overcome them, a new, homogeneous planner was developed. Future empirical studies have to reveal its performance.

References

[1] David Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32:333–377, 1987.

[2] Jürgen Dix, Joachim Posegga, and Peter H. Schmitt. Modal Logic for AI Planning. In *First International Conference on Expert Planning Systems*, pages 157–162, Brighton, GB, July 1990. IEE.

[3] G. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, 1969.

[4] Kave Eshghi. Abductive planning with event calculus. In *Proc. Fifth International Logic Programming Conference*, pages 562–579. MIT Press, 1988.

[5] R. Fikes and N. Nilsson. STRIPS: A new approach to theorem proving in problem solving. *Artificial Intelligence*, 2:189–205, 1971.

[6] Corell Green. Application of theorem proving to problem solving. In *1st International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, USA, 1969. appears also in Readings in Planning.

[7] Naresh Gupta and Dana S. Nau. Compexity results for blocks-world planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 629–633, San Francisco, USA, 1991. American Association of Artifical Intelligence.

[8] J.W. Lloyd. *Foundations Of Logic Programming (2nd Ed.)*. Springer, 1987.

[9] G. Moerkotte and P.C. Lockemann. Reactive consistency control in deductive databases. *ACM Trans. on Database Systems*, 16(4):670–702, 1991.

[10] G. Moerkotte, H. Müller, and J. Posegga. Aspects of consistency driven planning. In *Proc. 2nd. Int. Workshop on the Deductive Approach to Information Systems and Databases*, 1992.

[11] G. Moerkotte and P. Schmitt. Analysis and repair of inconsistencies in deductive databases. *submitted*, 1991.

[12] Guido Moerkotte. *Inkonsistenzen in deduktiven Datenbanken (Inconsistencies in deductive databases)*. Informatik-Fachberichte 248. Springer, Berlin, FRG, 1990.

[13] R. Reiter. On closed world data bases. *In: H. Gallaire And J. Minker (Eds.)*, Logic And Data Bases, Plenum, New York:227–253, 1978.

[14] Austin Tate. Generating project networks. In *5th International Joint Conference on Artificial Intelligence*, pages 888–893, Bosten, USA, 1977.

[15] E. Teniente and A. Olive. The events method for view updating in deductive databases. In *Proc. European Conf. on Extending Database Technology (EDBT)*, pages 245–260, 1992.