A New Logical Framework for Deductive Planning *

Werner Stephan and Susanne Biundo
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrucken
e-mail: <lastname>@dfki.uni-sb.cie

Abstract

In this paper we present a logical framework for defining consistent axiomatizations of planning domains. A language to define basic actions and structured plans is embedded in a logic. This allows general properties of a whole planning scenario to be proved as well as plans to be formed deductively. In particular, frame assertions and domain constraints as invariants of the basic actions can be formulated and proved Even for complex plans most frame assertions are obtained by purely syntactic analysis In such cases the formal proof can be generated in a uniform way. The formalism we introduce is especially useful when treating recursive plans A tactical theorem prover, the *Karlsruhe Interactive Verifier* KIV is used to implement this logical framework

1 Introduction

In this paper we present a logical framework for defining consistent axiomatizations of planning domains. An effective mechanism for defining the basic operations in a constructive way is embedded in a logic that allows properties of state spaces given by these actions to be proved. This not only includes the deductive generation of plans; reasoning about the complete scenario is also supported by this approach

Our work follows the *"Plans are Programs"* paradigm. This approach is not new; in *deductive* planning, in particular, it has already been discussed by several authors (cf. [Green, 1969; Rosenschein, 1981; Kautz, 1982; Bibel, 1986; Manna and Waldinger, 1987; Biundo *et* al., 1992]). However, their contributions concentrated mainly on aspects of common control structures like sequential composition, conditional branching, and sometimes recursion. Less attention has been paid to the question of data structures Our approach is based on the idea of treating relations used in the axiomatizations of planning scenarios in the same way data with an algebraic structure are treated in ordinary programming languages. In planning, situations are usually described by a set of relations between certain objects (blocks, rooms, robots etc.). These relations are flexible in the sense that they may change from one situation to another. The fact that these changes are *local* to a small section of the entire situation is not reflected in the basic semantic concepts underlying most formalisms for deductive planning. So, for example, the blocks world operation *unstack(a,b),* [Genesereth and Nilsson, 1987], changes only the on-relation between *a* and 6, the clear-property of 6 and the *table*-property of a If the pile of blocks is just a part of some room, which in turn is just one constituent of a larger scene, the unstack-operation exhibits very local behavior.

A straightforward concept is to consider these relations as objects, like elements of (abstract) data types in ordinary programming languages In the theory of abstract data types most often one only considers algebraic structures where all elements are (freely) generated by so-called constructors. In our case, restricting ourselves to finite relations, an appropriate set of Constructors" and "selectors" can easily be devised as well Starting with the empty relation all finite n-ary relations can be generated by successive applications of an *add-* operation that adds an n-tuple to a given relation As is the case with freely generated data types, we have to supply a corresponding *delete-operation* in order to compute with this data type It seems reasonable to take these two operations as the basis of a planning language designed to compute changes of relational structures. In our approach relations between unstructured objects correspond to data objects and are therefore considered to be finite. This seems to be a realistic assumption in most applications. In the context of *recursive* plans the finiteness of relations and its reflection by axioms becomes essential for *termination* proofs.

This approach, whereby one considers a fixed state space generated by *add-* (and *delete')* operations, underlies STRIPS-like [Fikes and Nilsson, 1971] planners. These systems also demand a complete description of the operations used in the planning process. In particular, they allow an efficient treatment of the *frame problem* [McCarthy and Hayes, 1969] since it is explicitly denoted which parts of a situation are changed. Our approach lends formal semantics to the basic concepts

*This work was partly supported by the German Ministry for Research and Technology (BMFT) under contract ITW 9000 8.

underlying the STRIPS approach and, while retaining its effectiveness, removes most of its limitations by providing a more general mechanism for defining operations and by embedding this mechanism in a logical framework. This not only extends the STRIPS approach and makes it suitable for deductive planning but also allows whole planning environments to be set up in a provably consistent way.

Based on elementary *add-* and delete-operations with fixed semantics, we will begin with the definition of basic actions, and then use efficient control structures to build up more complex plans. Apart from the usual ones, these control structures include a new non-deterministic choose construct, necessary to select objects in a non-deterministic way. *Domain constraints* are formulated as *tnvariants* of the basic actions and can be proved from their basic definitions, thus guaranteeing consistency of the whole planning environment. Due to the fixed semantics of the elementary operations, as well as the control structures, "side effects" can be excluded in many cases, by a purely *syntactical* inspection of plans, thereby providing an efficient treatment of the frame problem.

We will restrict ourselves to a formalization within a variant of Dynamic Logic (DL), although many of the basic ideas could be used in the context of other programming logics as well (e.g., [Salwicki, 1977; Harel, 1979; Manna and Pnueli, 1991; Kroger, 1987]). Not only does DL seem to be expressive enough for most applications in planning; this choice also allows an easy implementation of our formalism in an existing deductive system, the Karlsruhe Interactive Verifier (KIV) [Heisel *et al.*, 1990]. This system can be used as a *logic-based shell* for setting up planning environments. This includes the definition of the basic actions, the proof of invariants and additional lemmmata, and on top of that the implementation of various planning strategies

The paper is organized in the following way: section 2 introduces the semantic background of our theory. In section 3, we define the logic, including syntax and semantics of the planning language. Section 4 shows how state invariants can be derived from the basic operator definitions; it also shows how the abstract operator descriptions, already seen in other planning formalisms, can be obtained, and serve as the basis for deductive planning Section 5 is devoted to the frame problem In section 6 we discuss some aspects of the implementation within the KIV system. Section 7 refers to related work and finally, we conclude with some remarks in section 8.

2 State Spaces

Our logical language will be parameterized by an alphabet of "user-defined" symbols. Since many different sorts of objects occur in most applications, our *syntax* is based on a finite set of *sort symbols* Z. There will be a family of finite, disjoint sets, $\mathcal{C} = (C_z \mid z \in Z)$, the system of *constants*, and a family of denumerable, disjoint sets $\mathcal{X} = (X_z \mid z \in Z)$, the system of *variables*. Both families are assumed to be disjoint. Atomic formulae are built up by the equality symbol "$\equiv$" and "user-defined" *relation symbols* (e.g., *on*, *clear*, *table* etc.). These are given by a Z^*-indexed family $\mathcal{R} = (R_{\bar{z}} \mid \bar{z} \in Z^*)$ of disjoint sets, $R_{\bar{z}}$ being the set of relation symbols of *type* $\bar{z}$. We assume that almost all sets $R_{\bar{z}}$ are empty and that all of them are finite, i.e., $R := \bigcup\{R_{\bar{z}} \mid \bar{z} \in Z^*\}$ is a finite set.

For a given triple $(Z, \mathcal{R}, \mathcal{C})$ a *model* (defining the state space) is denoted by a structure $\mathcal{K} = (\mathcal{D}, S, \mathcal{I})$, where $\mathcal{D} = (D_z \mid z \in Z)$ is a system of *carrier sets*, S is a set of *states* (or situations) and $\mathcal{I}$ is a state-dependent *interpretation* that assigns an element of the appropriate carrier to each pair (s, c) and a relation of appropriate type to each pair (s, r). That is, for $r \in R_{\bar{z}}$ and $\bar{z} = (z_1, \ldots, z_n)$, we have $\mathcal{I}(s, r) \subseteq D_{z_1} \times \ldots \times D_{z_n}$.

In order to evaluate terms and formulae containing variables, we introduce *valuations* $\beta : X \to D$ that preserve sorts. Terms are either variables or constants. They are evaluated by $[\![c]\!]_{s,\beta} = \mathcal{I}(s, c)$ and $[\![x]\!]_{s,\beta} = \beta(x)$, respectively. Satisfiability for first-order formulas in structures $\mathcal{K}$ is defined by

$$\mathcal{K} \models_{s,\beta} r(\tau_1, \ldots, \tau_n) \text{ iff } ([\![\tau_1]\!]_{s,\beta}, \ldots, [\![\tau_n]\!]_{s,\beta}) \in \mathcal{I}(s, r),$$
$$\mathcal{K} \models_{s,\beta} \tau_1 \equiv \tau_2 \text{ iff } [\![\tau_1]\!]_{s,\beta} = [\![\tau_2]\!]_{s,\beta}, \text{ and}$$
$$\mathcal{K} \models_{s,\beta} \forall x \varphi \text{ iff for all } d \in D_z \ \mathcal{K} \models_{s,\beta^d_x} \varphi,$$

where $x \in X_z$ and β^d_x is like β except that $\beta^d_x(x) = d$.

As mentioned in the introduction, we are interested in particular finitely generated state spaces given by so-called *natural models* $\mathcal{K}_0 = (\mathcal{D}_0, S_0, \mathcal{I}_0)$. $\mathcal{D}_0$ is a family of at most countable sets. We define S_0 to be the set of all mappings that map the relation symbols from R to *finite* relations on $D_0 := \bigcup\{D_{0_z} \mid z \in Z\}$ of appropriate type and set $\mathcal{I}_0(s, r) = s(r)$. In natural models, constants are interpreted in a state independent way, that is for some d we have $\mathcal{I}_0(s, c) = d$ for all $s \in S_0$. Note that in natural models $\mathcal{K}_0$ we may drop the first index when evaluating terms.

The reason for introducing states was, of course, that we want to study operators that take us from one world to another. In particular, we are interested in a small set of operators that can be used as atomic constructs in our planning language, just as assignment statements are used in conventional programming languages. Looking at the state space given by a natural model $\mathcal{K}_0$ it is more or less obvious how these elementary operations should look. They can be defined for arbitrary structures $\mathcal{K}$ if treated as *relations* on S.

For each $r \in R_{\bar{z}}$ and $\bar{z} = (z_1, \ldots, z_n)$ let $d - r$ and $a - r$

$$\ldots - r : D_{z_1} \times \ldots \times D_{z_n} \to S \times S$$

be defined as

$$s \ d - r(d_1, \ldots, d_n) \ s' \text{ iff}$$
$$\mathcal{I}(s', r) = \mathcal{I}(s, r) - \{(d_1, \ldots, d_n)\} \text{ and}$$
$$\mathcal{I}(s', r') = \mathcal{I}(s, r') \quad \text{for } r' \neq r$$

and

$$s \ a - r(d_1, \ldots, d_n) \ s'' \text{ iff}$$
$$\mathcal{I}(s'', r) = \mathcal{I}(s, r) \cup \{(d_1, \ldots, d_n)\}, \text{ and}$$
$$\mathcal{I}(s'', r') = \mathcal{I}(s, r') \quad \text{for } r' \neq r$$

Theorem 1
1) In natural models $\mathcal{K}_0$ the relations $a - r(\ldots)$ and $d - r(\ldots)$ are total functions.

2) In natural models $\mathcal{K}_0$ for any two states s and s' there exists a finite sequence of elementary add -and delete-operations $op_1, \ldots, op_n$, such that $s\ op_1 \circ \ldots \circ op_n\ s'$, where '$\circ$' denotes the composition of relations.

3) If for elementary operations op_1, op_2

$\{op_1, op_2\} \neq \{d\text{-}r(d_1, \ldots, d_n), a\text{-}r(d_1, \ldots, d_n)\}$, *then*

$$op_1 \circ op_2 = op_2 \circ op_1 ,$$

$$d\text{-}r(d_1, \ldots, d_n) \circ a\text{-}r(d_1, \ldots, d_n) = a\text{-}r(d_1, \ldots, d_n) ,$$

$$a\text{-}r(d_1, \ldots, d_n) \circ d\text{-}r(d_1, \ldots, d_n) = d\text{-}r(d_1, \ldots, d_n) .$$

□

In general structures $\mathcal{K}$, if $\lambda x.\mathcal{I}(s,x) = \lambda x.\mathcal{I}(s',x)$ implies $s = s'$, the relations $a - r(\ldots)$ and $d - r(\ldots)$ are partial functions. A more serious restriction is imposed by requiring the *add-* and *delete*-operations to be total functions. This means that no interesting domain constraints will hold for the *whole* state space. But what can we do then if we are interested only in states where, for example, we have $on(y,x) \wedge on(z,x) \rightarrow y \equiv z$? The answer is simply to check whether the set of states with the above property is closed under the basic actions we want to use in our plans. That is, we treat domain constraints as *invariants* of the basic actions. Since the concept of invariant is broader than the concept of domain constraint—domain constraints must, of course, be invariants of the basic actions—our approach does not impose any restriction on the formulation of planning problems. It offers, as we shall discuss in section 4, the advantage of proving formulae to be invariant, thereby ensuring the soundness of the whole axiom system.

It can be proved (in a way which would have to be made precise) that all structures $\mathcal{K}$ where the constants are rigid symbols and $\mathcal{I}(s,r)$ is always a finite relation are "contained" in a natural model $\mathcal{K}_0$. Hence, the only crucial question is whether we can define (or "program") sufficiently many actions in a sufficiently abstract way. The property stated in part two of the theorem guarantees that the basis of our planning language is powerful enough: All states can be reached from each other by applying finite sequences of basic *add-* and *delete*-operations. Of course, this does not mean that all actions can be "programmed" in a satisfactory way. Indeed it turns out that the control structures have to be carefully designed. In particular, we need a non-deterministic *choose* construct in connection with recursive actions. The *choose* construct will be introduced in section 3.

3 The Logic

We start with the definition of the syntax of our planning language. Actions π and action abstractions γ are defined relative to a vocabulary given by $(Z, \mathcal{C}, \mathcal{R}, \mathcal{X})$. In addition, we use a system $\mathcal{A} = (A_{\bar{z}} \mid \bar{z} \in Z^*)$ of names for abstractions.

$$\begin{array}{lll} \pi & ::= & \textbf{skip} \mid \textbf{abort} \mid \textbf{delete-r}(\tau_1, \ldots, \tau_n) \mid \\ & & \textbf{add-r}(\tau_1, \ldots, \tau_n) \mid (\pi_1; \pi_2) \mid (\pi_1\ \textbf{or}\ \pi_2) \mid \\ & & \textbf{if}\ \varphi\ \textbf{then}\ \pi_1\ \textbf{else}\ \pi_2\ \textbf{fi} \mid \\ & & \textbf{choose}\ x\ \textbf{begin}\ \pi\ \textbf{end} \mid \gamma(\tau_1, \ldots, \tau_n) \\ \gamma & ::= & a \mid \textbf{rec}\ a(x_1, \ldots, x_n).\pi \mid \textbf{rec}_n a(x_1, \ldots, x_n).\pi, \end{array}$$

where φ is a first-order formula and r is a relation symbol from R. We impose the usual type constraints. The occurrence of x following **choose** is a binding occurrence, the scope of which consists of the plan enclosed by "**begin**" and "**end**".

Let $\mathcal{K} = (\mathcal{D}, S, \mathcal{I})$ be a structure for $(Z, \mathcal{C}, \mathcal{R})$. The semantics of plans π is given by a valuation $[\![\ldots]\!]_\beta$, where $[\![\pi]\!]_\beta \subseteq S \times S$.

$[\![\textbf{skip}]\!]_\beta = \{(s,s') \mid s = s'\}$
$[\![\textbf{abort}]\!]_\beta = \{\}$
$[\![\ldots r(\tau_1, \ldots, \tau_n)]\!]_\beta =$
$\quad \{(s,s') \mid s \in \ldots r([\![\tau_1]\!]_{s,\beta}, \ldots, [\![\tau_n]\!]_{s,\beta})\ s'\}$
$[\![(\pi_1; \pi_2)]\!]_\beta = [\![\pi_1]\!]_\beta \circ [\![\pi_2]\!]_\beta$
$[\![\textbf{if}\ \varphi\ \textbf{then}\ \pi_1\ \textbf{else}\ \pi_2\ \textbf{fi}]\!]_\beta =$
$\quad \Big((\varphi?)_\beta \circ [\![\pi_1]\!]_\beta\Big) \cup \Big((\neg\varphi?)_\beta \circ [\![\pi_2]\!]_\beta\Big),$
$\quad$ where $s\ (\varphi?)_\beta\ s'$ iff $s = s'$ and $\mathcal{K} \models_{s,\beta} \varphi$
$[\![(\pi_1\ \textbf{or}\ \pi_2)]\!]_\beta = [\![\pi_1]\!]_\beta \cup [\![\pi_2]\!]_\beta$
$[\![\textbf{choose}\ x\ \textbf{begin}\ \pi\ \textbf{end}]\!]_\beta = \bigcup \{[\![\pi]\!]_{\beta^x_d} \mid d \in D_x\},$
$\quad$ where $x \in X_x$
$[\![\textbf{rec}\ a(x_1, \ldots, x_n).\pi\ (\tau_1, \ldots, \tau_n)]\!]_\beta =$
$\quad \bigcup \{[\![\textbf{rec}_n a(x_1, \ldots, x_n).\pi\ (\tau_1, \ldots, \tau_n)]\!]_\beta \mid n \geq 0\}$
$[\![\textbf{rec}_{n+1} a(x_1, \ldots, x_n).\pi\ (\tau_1, \ldots, \tau_n)]\!]_\beta =$
$\quad [\![\pi^*(a \leftarrow \textbf{rec}_n a(x_1, \ldots, x_n).\pi)]\!]_{\beta^{d_1 \ldots d_n}_{x_1 \ldots x_n}},$
$\quad$ where $d_i = [\![\tau_i]\!]_\beta$ for $1 \leq i \leq n$ and π^* results from π by suitably renaming the bound variables, so as to avoid clashes.
$[\![\textbf{rec}_0\ a(x_1, \ldots, x_n).\pi\ (\tau_1, \ldots, \tau_n)]\!]_\beta = \{\},$
$[\![a(\tau_1, \ldots, \tau_n)]\!]_\beta = \{\}$

We have defined the semantics for arbitrary structures $\mathcal{K}$. However, the reader should bear in mind that we are interested only in natural models $\mathcal{K}_0$, where the *add-* and *delete*-operations are total functions on the set of states.

rec $a(x_1, \ldots, x_n).\pi$ is a recursive action $x_1, \ldots, x_n$ being the formal parameters and π being the body of that action. To simplify our exposition we restrict ourselves to simple recursive actions, an extension to mutually recursive ones being straightforward. The semantics (and proof theory) of recursive abstractions relies on *finite approximations*. As can be seen from the semantic definitions above **rec**$_n$ $a(x_1, \ldots, x_n).\pi$ denotes the n'th approximation of the meaning of **rec** $a(x_1, \ldots, x_n).\pi$. *Uninterpreted reasoning* is achieved by using induction on the indices of approximations in order to prove statements about recursive actions.

The language is *referentially transparent* with respect to variables. Side effects occur only on the level of relations (states). This is reflected in the axiomatization which, in this aspect, is simpler than that of ordinary DL and close to that given in [Kautz, 1982].

The *choose* construct guesses a new element by changing the valuation (environment) that is used to evaluate the subsequent action. However, this change of environments as in the case of parameter passing follows a strict stack discipline, that is, the effect of choosing a new element can not work outside the plan enclosed by "**begin**" and "**end**". The *choose* construct is necessary to "move" in structures.

To be able to reason about these changes, we follow the approach taken in Dynamic Logic, in that we extend the predicate logic used so far by formulae $[\pi]\varphi$ and $[?-$

$r(\xi_1, \ldots, \xi_n)]\varphi$, where r is a relation symbol and the ξ_i are either terms or "placeholders" $\ominus$.

The formal semantics of this new type of formulas is given by

$$\mathcal{K} \models_{s,\beta} [\pi]\varphi \text{ iff } \mathcal{K} \models_{s',\beta} \varphi \text{ for all } s' \text{ such that } s\ [\![\pi]\!]_\beta\ s'$$

$$\mathcal{K} \models_{s,\beta} [? - r(\xi_1, \ldots, \xi_n)]\varphi \text{ iff } \mathcal{K} \models_{s',\beta} \varphi \text{ for all } s' \text{ such that } \mathcal{I}(r, s') \text{ and } \mathcal{I}(r, s) \text{ differ in } (d_1, \ldots, d_n), \text{ only if for all } 1 \le i \le n\ \ d_i = [\![\xi_i]\!]_\beta \text{ or } \xi_i = \ominus \text{ and } \mathcal{I}(r', s') = \mathcal{I}(r', s) \text{ for all } r' \neq r$$

Intuitively, $[\pi]\varphi$ has to be read: *"If π terminates, φ holds afterwards."* The dual operator $<\pi>$, defined by $<\pi>\varphi :\leftrightarrow \neg[\ \pi\ ]\neg\varphi$ has to be read: *"π terminates with φ."* The (modal) operator $[? - r(\xi_1, \ldots, \xi_n)]$ refers to all states that differ from the given one in at most the value of r, where some argument positions are fixed. For example, $[? - on(\ominus, \ominus)]\ \varphi$ means that φ holds in all states that differ from the given one in at most the *on*-relation whereas $[? - on(\tau_1, \tau_n)]\varphi$ means that φ holds in all states that differ from the given one in at most the *on*-relation between the objects denoted by τ_1 and τ_n. In ordinary DL, this is achieved by quantifying on program variables. A kind of "quantification" like the one above increases the expressive power of the formalism in general and is *necessary* for inductive proofs.

The axiomatization follows the paradigm of so-called *uninterpreted* reasoning, where we do not rely on the expressive power of the underlying data structure. The semantics and proof theory of recursive actions (plans) is outside the scope of this paper, however, a general introduction to uninterpreted reasoning, as it is implemented in the KIV system, can be found in [Heisel *et al.*, 1989]. An axiomatization of a very powerful procedure concept for imperative programming languages is given in [Stephan, 1989]. Here we present only some axioms for the non-standard constructs.

As is the case with assignments in ordinary programming languages, the effects of the *add-* and *delete-*operations on first-order formulae can be described in an exhaustive way.

Theorem 2 *Let φ be a formula such that all bound variables are distinct from the variables occurring in $\tau_1, \ldots, \tau_n$. Then the weakest precondition (in the sense of* [Dijkstra, 1976]*) of φ with respect to* **delete-r**$(\tau_1, \ldots, \tau_n)$ *and* **add-r**$(\tau_1, \ldots, \tau_n)$ *are the formulae $\hat{\varphi}$ and $\check{\varphi}$, respectively, where $\hat{\varphi}$ results from φ by replacing all atomic subformulae*

$r(\sigma_1, \ldots, \sigma_n)$ *by* $\Big(r(\sigma_1, \ldots, \sigma_n) \wedge (\tau_1 \not\equiv \sigma_1 \vee \ldots \vee \tau_n \not\equiv \sigma_n)\Big)$

and $\check{\varphi}$ results from φ by replacing all atomic subformulae

$r(\sigma_1, \ldots, \sigma_n)$ *by* $\Big((\tau_1 \not\equiv \sigma_1 \vee \ldots \vee \tau_n \not\equiv \sigma_n) \rightarrow r(\sigma_1, \ldots, \sigma_n)\Big)$.

□

Analogous to the well known assignment axioms we have

$$[\textbf{delete-r}(\tau_1, \ldots, \tau_n)]\varphi \leftrightarrow \hat{\varphi} \text{ and}$$
$$[\textbf{add-r}(\tau_1, \ldots, \tau_n)]\varphi \leftrightarrow \check{\varphi}\ .$$

In addition to that, uninterpreted reasoning requires axioms like:

$$\bigvee(x_i \not\equiv y_i \mid 1 \le i \le n) \rightarrow [\textbf{delete-r}(\bar{x})][\textbf{add-r}(\bar{y})]\ \varphi \leftrightarrow [\textbf{add-r}(\bar{y})][\textbf{delete-r}(\bar{x})]\ \varphi\ ,$$
$$[\textbf{delete-r}(\bar{x})][\textbf{add-r}'(\bar{y})]\ \varphi \leftrightarrow [\textbf{add-r}'(\bar{y})][\textbf{delete-r}(\bar{x})]\ \varphi\ ,$$
$$[\textbf{delete-r}(\bar{x})][\textbf{add-r}(\bar{x})]\ \varphi \leftrightarrow [\textbf{add-r}(\bar{x})]\ \varphi\ ,$$
$$[\textbf{add-r}(\bar{x})][\textbf{delete-r}(\bar{x})]\ \varphi \leftrightarrow [\textbf{delete-r}(\bar{x})]\ \varphi,$$

For the choose construct we have

$$[\textbf{choose } x \textbf{ begin } \pi \textbf{ end}]\varphi \leftrightarrow \forall y.[\pi^y_x]\varphi\ ,$$

where y is a fresh variable. The axioms for the simple structured commands are as usual. Examples for general (modal) axioms are

$$[\pi](\varphi \rightarrow \psi) \rightarrow ([\pi]\varphi \rightarrow [\pi]\psi)\ ,$$
$$\forall x[\pi]\varphi \rightarrow [\pi]\forall x\varphi\ ,$$

where x must not occur free in π, and

$$\forall x.\varphi \rightarrow \varphi^\tau_x\ ,$$

where τ is free for x in φ, and φ^τ_x denotes the substitution of τ for all free occurrences of x in φ.

4 Actions and Plans

We are now going to outline how planning domains can consistently be defined in our theory. In section 6 we will discuss briefly the technical aspects of a logic based shell for planning and the implemention of planning strategies within such a system. Our treatment of the frame problem will be discussed seperately. The main concern of this section is to demonstrate how basic actions can be defined in an *abstract* way and that domain constraints can be treated adequately.

There is no technical distinction between basic actions and derived plans composed out of them. The latter can be used without restriction as basic operations for higher levels of the planning process. However, we have to start out with some set of basic actions that are hand-coded. The first step involves fixing the set of relation symbols we want to use. At the end of this section we will discuss how defined notions, like *above* in the blocks world scenario can be added to the theory.

The *unstack* operation, for example, can be defined as a simple abstraction γ_{un}.

$$\begin{array}{lll} \textbf{rec } \mathit{unstack}(x, y). & \textbf{if } \mathit{on}(x, y) \wedge \mathit{clear}(x) & \\ & \textbf{then } \textbf{add} - \mathit{table}(x); & \\ & \quad\ \textbf{add} - \mathit{clear}(y); & \\ & \quad\ \textbf{delete} - \mathit{on}(x, y) & \\ & \textbf{else } \textbf{abort } \textbf{fi}\ . & \end{array}$$

In most applications, the state space is restricted by so-called *domain constraints*. In the blocks world, for example, we have $\forall x\ \Big(\mathit{clear}(x) \leftrightarrow \neg\exists y\ \mathit{on}(y, x)\Big)$. In our setting, domain constraints are treated as *invariants*. For φ being the equivalence above, we can *prove* the assertion $\varphi \rightarrow [\gamma_{un}(x, y)]\varphi$. If a similar assertion

holds for all basic actions, we can use the domain constraint φ in all states reached by arbitrary plans made up of these basic actions, provided φ has been included in the description of the initial state. This fact can be proved formally in our setting.

The great advantage is that adding domain constraints in such a way guarantees *consistency* with the definition of the basic actions and, with that, consistency of the whole planning environment. Having deduced a set of domain constraints, we may also simplify the *description* of the basic actions to be used in the planning process. From the definition given above, a sufficient (abstract) description of the *unstack* operation (in the style of [Kautz, 1982]) would be:

$$on(x,y) \wedge clear(x) \rightarrow \ <\gamma_{un}(x,y)> table(x) \wedge clear(y).$$

As another example, suppose we have a world where blocks have colours and let *red*, *black*, and *white* be the only colours that occur. An operator *paintblack* [Bibel *et al.*, 1989] that changes the colour of any block to black is defined by the abstraction γ_{pb}

rec *paintb*(x). **add** $-$ *black*(x);
delete $-$ *white*(x);
delete $-$ *red*(x) .

If we can prove that the formula
$\forall x \ \Big(red(x) \dot{\vee} black(x) \dot{\vee} white(x)\Big)$, stating that every block has exactly one of the three colours, is an invariant of all basic actions, then $<\gamma_{pb}(x)> black(x)$ sufficiently describes our action.

Note that it is not necessary to specify adding or deleting *negative* facts in our approach. So, even without the domain constraint above, $\neg white(x)$ as well as $\neg red(x)$ can be proved to hold after the execution of γ_{pb}.

Definitions of basic actions can be more complex than those presented above. Using the *choose* construct we are able to define, for example, the non-deterministic *dump* operator [Kautz, 1982], that transfers *all* blocks from a certain box into another by

rec *dump*(x,y). **if** $\exists\ z\ in(z,x)$
then choose z
begin if $in(z,x)$
then add $-$ $in(z,y)$;
delete $-$ $in(z,x)$;
dump(x,y)
else abort fi
end
else skip fi

and prove theorems about it.

The language introduced above can also be used to treat "recursively defined notions", like the *above* relation in the blocks world scenario. This kind of relations often causes problems in planning environments, see for example [Kautz, 1982]. Let γ_{ab} be the recursive abstraction

rec *above*(x,y). **if** $\neg on(x,y)$
then choose z
begin if $on(z,y)$
then *above*(x,z)
else abort fi
end
else skip fi .

Using this piece of program the relation *above* can be defined by

$$above(x,y) \ :\leftrightarrow \ <\gamma_{ab}(x,y)>\mathbf{true}$$

We are then able to prove lemmata like

$\Big(\bigwedge \Sigma \wedge above(x,y) \wedge x \not\equiv u\Big) \rightarrow [\gamma_{un}(u,v)]above(x,y)$,
$on(x,y) \rightarrow above(x,y)$, and
$\Big(on(y,z) \wedge above(x,y)\Big) \rightarrow above(x,z)$,

where Σ is the set of domain constraints.

5 The Frame Problem

Considering the *unstack*-operation and *analyzing* its definition reveals the fact that the only relations affected by *unstack* are *table*, *clear*, and *on*. This, in particular, means that *unstack* has no side effects on, for example, the colours of blocks. Thus,

$$white(x) \ \rightarrow \ [\gamma_{un}(a,b)]white(x)$$

appears to be a valid (*frame*) *assertion*. This observation leads us to an efficient treatment of the frame problem, which has the following proper foundation.

In our approach, the basic actions of a planning domain are *defined* as abstractions, the bodies of which in simple cases merely contain the elementary *add*- and *delete*-operations. From these definitions of basic actions, frame assertions can be *inferred* using uninterpreted reasoning. Clearly, we have to use the basic axioms for the *add*- and *delete*-operations, respectively, in these deductions. In general, frame assertions are of the form

$$r(\tau_1,\ldots,\tau_n) \rightarrow [\pi]\Big(\epsilon \rightarrow r(\tau_1,\ldots,\tau_n)\Big) \ ,$$

where the condition ϵ consists of inequalities. In fact, another frame assertion for the *unstack*-operation would be:

$$on(x,y) \rightarrow [\gamma_{un}(a,b)]\Big((x \not\equiv a \vee y \not\equiv b) \rightarrow on(x,y)\Big) \ .$$

Of course, the frame assertions can be proved for complete plans as well.

One main advantage of our approach, however, is that a comprehensive subset of valid frame assertions can be obtained in a non-deductive way by an algorithm that analyzes the syntactical structure of plans. Assertions generated by this algorithm can be *proved* in a *uniform* way, that is, we can provide a proof procedure (tactic in KIV) that automatically generates a proof for each such assertion. We shall now outline the basic ideas underlying this algorithm.

In order to formulate the general method for computing sound frame assertions (for general plans), we have

to analyze the semantics of our planning language. It turns out that for each plan π and each relation symbol r, the formal execution tree of π contains only a finite number of different applications of the elementary operations $.. - r(\ldots)$, if we do not take into account those arguments that are program variables bound by a *choose* construct. In such a generalized 'call' $.. - r(\hat{\tau}_1, \ldots, \hat{\tau}_n)$, we write $\hat{\tau}_i = \odot$, if $\hat{\tau}_i$ is such an argument. Let $R_d(\pi, r)$ and $R_a(\pi, r)$ denote the set of all applications of elementary *add*- and *delete*-operations that are reachable by π, respectively. Using this notation we get the following result.

Theorem 3 *For each π and each relation symbol r the following implications are provable in our axiomatization :*

$$r(\sigma_1, \ldots, \sigma_n) \rightarrow [\pi] \bigwedge(cond(op) \mid op \in R_d(\pi, r)) \rightarrow r(\sigma_1, \ldots, \sigma_n) ,$$
$$\neg r(\sigma_1, \ldots, \sigma_n) \rightarrow [\pi] \bigwedge(cond(op) \mid op \in R_a(\pi, r)) \rightarrow \neg r(\sigma_1, \ldots, \sigma_n) ,$$
$$\text{where } cond(op) = \bigvee(\hat{\tau}_i \not\equiv \sigma_i \mid \hat{\tau}_i \neq \odot \text{ and } 1 \leq i \leq n) .$$

Clearly, the extensive use of the *choose* construct reduces the number of computed frame axioms. All cases not covered by these computed frame axioms have to be proved in a non-uniform way.

6 Implementation

Although this paper is mainly devoted to the presentation of our theory, we will shortly describe how a logic based planning environment can be implemented within an (existing) tactical theorem proving system. The paradigm of tactical theorem proving seems to be especially well suited to the kind of environment we have in mind (see also [Guinchiglia *et al.*, 1992]). Based on a general logical framework, derived rules and tactics can be defined and are then used to implement efficient planning strategies or other reasoning methods on plans.

Like many other systems in the area of tactical theorem proving the KIV system is based on a *sequent calculus*. Program (plan) synthesis, [Heisel *et al.*, 1991], is supported by so-called meta variables. Given planning problems by sets of formulae Γ and Δ, we start with the goal $\Sigma, \Gamma \Rightarrow <?a> \bigwedge \Delta$, and instantiate the metavariable $?a$ during a goal-directed (backward-chaining) proof. Σ is the set of domain constraints. Strategies like *progression* and *regression* (cf. [Kautz, 1982]) can easily be implemented on the basis of a set of suitable derived rules and tactics. For example, our treatment of domain constraints can be implemented by a derived scheme like $\Sigma \Rightarrow [\pi] \bigwedge \Sigma$. These strategies can use the "computed" frame assertions to determine the invariant part of a pre- or postcondition.

In this way, we obtain considerable efficient implementations that can be easily changed, extended, and combined and that are guaranteed to be sound with respect to the basic formalism.

7 Related Work

Rosenschein and Kautz were the first using Dynamic Logic in planning [Rosenschein, 1981; Kautz, 1982]. They define basic actions as atomic constituents of their planning language that are axiomatized freely by describing their preconditions and effects, respectively. Our approach goes beyond this by providing a STRIPS-like way of defining basic actions and setting up consistent planning scenarios on top of that. The logical formalism is extended to reason about these basic actions as well as about composite plans built out of them. In both cases this includes recursive definitions.

With the work of Pednault [Pednault, 1986; Pednault, 1989] the approach we presented in this paper shares the idea of describing basic actions in a STRIPS-like manner, that is, by giving add and delete lists for relations. Moreover, both approaches embed these descriptions into a logical formalism that can be used to reason about plans. We begin with the observation that the appropriate semantical background for integrating this STRIPS approach into deductive planning are models based on finitely generated relations. While ADL uses a fixed form of conditional add and delete lists our approach allows to program basic operations in a carefully chosen programming language that covers ADL schemata in a straightforward way. In our approach it is easy to add non-determinism and also in the deterministic case we can do without auxiliary relations which seem to be necessary in ADL to describe more complicated actions. In our setting we start out with the *definition* of basic actions. The defining programs always have a precise meaning in the underlying semantical structures. From these definitions which in addition are independent of each other we then *prove* domain constraints, derived descriptions, and frame assertions. These issues are not addressed in the ADL work. In addition, we have outlined a method to generate certain frame assertions even for composite plans by a purely textual analysis. Although many ideas presented above are independent of the logical basis we want to stress that the ability to reason about the structure of (possibly) recursive definitions (programs) is essential in this context.

As already mentioned above, our formalism is based on the STRIPS ideas that have been given formal semantics by Lifschitz [Lifschitz, 1986]. We feel that our formalism in some sense "proceduralizes" Lifschitz's approach and extends it in some way, e.g. as far as the treatment of negative effects etc. is concerned. However, investigating this relationship in more detail goes beyond the scope of this paper. Separate work will be devoted to that issue.

8 Conclusion

Combining characteristic features of conventional planning with techniques borrowed from programming logics, we have introduced a new theory of action based on a special variant of Dynamic Logic. Plans may be constructed using rich control structures including recursion, non-deterministic branching, and a special *choose* construct. In our approach, we start out by the definition of basic actions and are then able to prove properties about the state space generated by these actions. This includes frame assertions and domain constraints. In this way, we prevent our planning environment from running into inconsistencies. These are possible in other

systems where frame assertions and domain constraints are considered to be axioms. An efficient treatment of the frame problem is provided by a method to generate most frame assertions non-deductively (with the possibility of a uniform formal proof within the system). Our theory of action clearly is not restricted to blocks-world-type planning domains. One could equally well define a theory for an intelligent help system context where the planning domain is a command language environment. There the ability of reasoning about recursive plans is essential. Implementing our logical framework in the KIV system provides the basis not only for a deductive planning system, but also for a complete deductive planning environment, i.e., a system that also assists a user in developing a consistent axiomatization of his planning domain. Furthermore, this notion of environment can be extended by implementing tactics for temporal projection, plan validation and other reasoning methods. Further work is devoted to the automated generation of recursive plans and an extension of the logical framework to parallelism.

References

[Bibel et a/., 1989] W. Bibel, L. Farinas del Oerro, B. Fronhofer, and A. Herzig. Plan Generation by Linear Proofs: On Semantics. In GWA189: Curman Workshop on Artificial Intelligence, pages 50-62 Springer LNCS 216, 1989.

[Bibel, 1986] W. Bibel. A Deductive Solution for Plan generation. New Generation Computing, 4:115-132, 1986.

[Biundo et al, 1992] S. Bitindo, D. Dengler, and J. Kohler. Deductive Planning and Plan Reuse in a Command Language Environment. In Proceedings of the 10th European Conference on Artificial Intelligence, pages 628-632, 1992.

[Dijkstra, 1976] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, London, 1976.

[Fikes and Nilsson, 1971] R E. Fikes and N..I Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence, 2:189-208, 1971.

[Genesereth and Nilsson, 1987] M R. Genesereth and N.J. Nilsson. Logical Foundations of Artificial Intelligence. Morgan Kaufmann Publishers, Los Altos, California, 1987.

[Green, 1969] C. Green. Application of Theorem Proving to Problem Solving. In Proceedings of the 1st International Joint Conference on Artificial Intelligence, pages 219-239, 1969.

[Guinchiglia et al, 1992] F. Guinchiglia, P. Traverso, A. Cimatti, and L. Spalazzi. Tactics. Extending the Notion of Plan. In Proc. of the ECA1-92 Workshop on Beyond Sequential Planning, 1992.

[Harel, 1979] D. Harel. First Order Dynamic Logic. Springer LNCS 68, New York, 1979.

[Heisel et al, 1989] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, Proceedings of Logic at Botik, pages 134-145. Springer LCNS 363, 1989.

[Heisel et al, 1990] M. Heisel, W. Reif, and W. Stephan. Tactical Theorem Proving in Program Verification. In Proceedings of the 10th International Conference on Automated Deduction, pages 117-131. Springer LCNS 449, 1990.

[Heisel et al, 1991] M. Heisel, W. Reif, and W. Stephan. Formal Software Development in the KIV System. In Automating Software Design, R. McCartney and M.R. Lowry (eds.). AAAI Press, 1991.

[Kautz, 1982] H.A. Kautz. Planning within First-Order Dynamic Logic. In Proceedings of the CSCSI/SCEIO, pages 19-26, 1982.

[Kroger, 1987] F. Kroger. Temporal Logic for Programs. Springer, Berlin, Heidelberg, New York, 1987.

[Lifschitz, 1986] V. Lifschitz. On the Semantics of STRIPS. In M P. Georgeff and A L. Lansky, editors, Reasoning about Actions and Plans, pages 1-8. Morgan Kaufmann Publishers, Los Altos, 1986.

[Manna and Pnueli, 1991] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer, New York, 1991.

[Manna and Waldinger, 1987] Z. Manna and R.. Waldinger How to Clear a Block: Plan Formation in Situational Logic. Journal of Automated Reasoning, 3:343 377, 1987.

[McCarthy and Hayes, 1969] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, Machine Intelligence Vol 4, pages 463-502. Edinburgh University Press, Edinburgh, 1969.

[Pednault, 1986] E. Pednault. Formulating Multiagent, Dynamic-World Problems in the Classical Planning Framework. In M P. Georgeff and A L. Lansky, editors, Reasoning about Actions and Plans, pages 47-82. Morgan Kaufmann Publishers, Los Altos, 1986.

[Pednault, 1989] E. Pednault. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning, pages 324 332. Morgan Kaufmann Publishers, 1989.

[Rosenschein, 1981] S. Rosenschein. Plan Synthesis: A Logic Perspective. In Proceedings of the 7th International Joint Conference on Artificial Intelligence, pages 331-337, 1981.

[Salwicki, 1977] A. Salwicki. Algorithmic Logic. A Tool for Investigations of Programs. In Logic, Foundations of Mathematics, and Computability Theory, Butts and Hintikka (eds.), pages 281-295. D. Reidel Publishing Company, Dordrecht, Holland, 1977.

[Stephan, 1989] W. Stephan. Axiomatisierung rekursiver Prozeduren in der Dynamischen Logik. Habilitationsschrift, Universitat Karlsruhe, Karlsruhe, 1989.