Chapter 1

FUNCTIONAL STRIPS: A MORE FLEXIBLE LANGUAGE FOR PLANNING AND PROBLEM SOLVING

Héctor Geffner
Departamento de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas 1080-A
hector@usb.ve

Abstract Effective planning requires good modeling languages and good algorithms. The Strips language has shaped most of the work in planning since the early 70's due to its effective solution of the frame problem and its support for divide-and-conquer strategies. In recent years, however, planning strategies not based on divide-and-conquer and work on theories of actions suggest that alternative languages can make modeling and planning easier. With this goal in mind, we have developed Functional Strips, a language that adds first-class function symbols to Strips providing additional flexibility in the codification of planning problems. This extension is orthogonal and complementary to extensions accommodated in other languages such as conditional effects, quantification, negation, etc. Function symbols, unlike relational symbols, can be nested so objects need not be referred to by their explicit names and as a result more efficient encodings can be provided. For example, a problem like the 8-puzzle can be codified in terms of four actions with no arguments; Hanoi, can be codified with a number of ground actions independent of the number of disks; resources and constraints can be easily represented, etc.

Functional Strips is both an action and a planning language in the sense that actions are understood declaratively in terms of a state-based semantics and operationally in terms of efficient updates on state *representations.* In this paper, we present the language, the semantics and a number of examples, and discuss possible uses in planning and problem solving.

1. INTRODUCTION

The Strips language introduced by Fikes and Nilsson in 1971 has shaped most the work in Planning (FN71). The appeal of Strips can be explained by two main reasons: first, it provides a compact representation of actions that avoids the frame problem, and second, it supports divide-and-conquer strategies that have been regarded as good for planning. Strips, however, is limited in several ways, and more recent work on action representation (Ped89; Rei91; GL93; San94; Sha97) and planning algorithms (e.g., (KS99; BG99)) suggests that other languages could facilitate both modeling and computation.

A number of extensions of the basic Strips language, including negation, conditional effects, state variables, and quantification have been considered in a number of proposals (e.g., (Wil88; Ped89; McD98b)). These extensions simplify the modeling task and in certain cases make plans shorter (Neb98). These planning languages, however, as well as the closely related action languages developed in the area of reasoning about change (in particular, (GL93; GKL97)) share with Strips an important restriction: the symbols that can be used to represent fluents are either constant or relational symbols (such as $\mathbf{on}(a, b)$ or **fluid_level**), but not function symbols. Functional symbols are excluded or they are accommodated with restrictions (e.g., they cannot appear nested in the head of action rules). Functional relations, however, are important in planning, and indeed, the recent PDDL language standard (McD98b) makes room for terms involving non-boolean fluents, but largely in an ad-hoc fashion.

The goal of this paper is extend Strips with first-class function symbols, generalizing and making explicit the logic underlying Strips-like languages.[1] In particular, we try to clarify the distinction between *state* and *state representations,* two notions that are often collapsed in planning. States are *logical interpretations* over the the language providing a denotation to all constant, function, and relational symbols. States *representations* on the other hand, are encodings of those interpretations whose form depends on the language. In the case of Strips, states can be represented by sets of atoms, while in other languages, they can be represented by sets of literals or suitable assignments. In all cases, the representations encode interpretations that determine the denotation of all relevant expressions in the language.

One of the main advantages of a functional language over a purely relational one is that functions can be nested and thus can refer to objects

[1] For an earlier logical account of Strips with a different scope, see (Lif86).

without providing their explicit names. As a result, the number of action arguments and the number of possible *ground actions* can be reduced substantially, something that is crucial in modern planners (BF95; KS99; BG99). For example, the 8-puzzle can be modeled in Functional Strips with 4 ground actions only, Hanoi can be modeled with a number of ground actions that is independent of the number of disks, and so on. In other problems, like Gripper (McD98a), the new language allows for entirely different formulations that exploit the high degree of symmetry in the domain (FL99). In many cases, the representations obtained are as efficient as the representations used in specialized programs, something that is necessary, although not sufficient, for planning approaches to be competitive with specialized methods.

Two additional comments before we proceed. Once function symbols become 'first-class citizens', the number of possible atoms in the language becomes infinite. This does not mean that state representations become infinite as well. Indeed, the *representation* of states, while different from Strips, remains finite and compact as a consequence of the assumption that the domain of interpretation is given by a *finite* set of objects. At the same time, the *computation* of the representation of the next state remains efficient as well.

The second issue is that while functions in Strips help reduce the number of possible *ground actions,* they do not and cannot affect the *branching factor* of the problem; namely, the number of actions that are applicable in each state. Still, it's worth emphasizing that the performance of modern planners including Graphplan, SAT, and heuristic-based planners (BF95; KS99; BG99) is influenced *both* by the branching factor of the problem *and* the number of possible ground actions. Even planners that rely on user-supplied control knowledge such as TLPLAN (BK98), are affected by the number of ground actions as they all must be tested for applicability in each state. In a Strips encoding of a problem like Hanoi, there may be up to N^3 ground actions, where N is the number of disks, but at most only 3 of these actions are applicable in each state. In Functional Strips, the number of ground actions for this problem can be reduced to 6, a number of actions that is closer to the branching factor of the problem and can more efficiently be tested at run time.[2]

[2]Models that reduce the number of possible ground actions towards the limit represented by the branching factor of the problem are likely to be more suitable for *learning* as well. Indeed, learning to apply actions that take few arguments is likely to be simpler than than learning to apply actions that take many arguments. This is true for example in (Kha97), where general action strategies are learned for general domains expressed in Strips.

The rest of the paper is organized as follows. We review Strips and State Models (Sect. 2), show how to accommodate functions in Strips (Sect. 3), and illustrate the resulting language over a number of examples (Sect. 4). We also illustrate the importance of choosing suitable representations by considering the so-called Gripper domain (Section 5). We then discuss possible uses of Functional Strips in actual planning systems (Sect. 6), and briefly consider a number of additional extensions and related work (Sect. 7).

2. STRIPS

2.1 LANGUAGE

The Strips language comprises two parts: a language for describing the world and a language for describing how the world changes. The first is called the *state language* or simply the language, and the second, is called the *operator language.* We consider the Strips language as used currently in planning (e.g., the Strips subset of PDDL (McD98b)), rather the original version of Strips that is slightly more complex (FN71; Lif86).

The Strips language $\mathcal{L}_S$ is made up of two types of symbols: relational and constant symbols. In the expression $on(a, b)$, on is a relational symbol of arity 2, and a and b are constant symbols. We refer to the (finite) sets of relational and constant symbols as $\mathcal{R}$ and $\mathcal{C}$ respectively. In the Strips language, there are no functional symbols and the constant symbols are the only *terms.* The *atoms* are defined in a standard way from the combination $p(t_1, \ldots, t_k)$ of a relational symbol p and a tuple of terms t_i of the same arity as p. Similarly the Strips *formulas* are obtained by closing the set of atoms under the standard propositional connectives. In Strips, only conjunctions are used and they are identified with sets of atoms.

A main difference between relational and constant symbols in Strips is that the former are used to keep track of aspects of the world that may change as a result of the actions (e.g., $\mathbf{on}(a, b)$), while the latter are used to refer to objects in the domain (e.g., $on(\mathbf{a}, \mathbf{b})$). More precisely, actions in Strips affect the denotation of relational symbols but not the denotation of constant symbols. For this reason, the former are said to be *fluent* symbols, and the latter, *fixed* symbols or *constants.*

The *operators* are defined over the set of atoms $\mathcal{A}$ in $\mathcal{L}_S$. Each operator op has a precondition, add, and delete lists $Prec(op)$, $Add(op)$, and $Del(op)$ given by sets of atoms. Operators are normally defined by means of *schemas;* here we assume that such schemas have been grounded.

A Strips planning problem $P = \langle \mathcal{L}_S, \mathcal{O}_S, \mathcal{I}_S, \mathcal{G}_S \rangle$ consists of a tuple where $\mathcal{L}_S$ stands for the state language (defined by the constant and relational symbols), $\mathcal{O}_S$ is the set of operators defined over the atoms in $\mathcal{L}_S$, and $\mathcal{I}_S$ and $\mathcal{G}_S$ are sets of atoms defining the *initial* and *goal* situations.

2.2 STATE MODELS

The meaning of a Strips planning problem (as well as the meaning of problems expressed in many extensions of Strips) can be given in terms of *state-space models* (NS72; Nil80; Pea83). A state model is a tuple $\langle S, s_0, S_G, A, next \rangle$ where

1. S is a finite set of states
2. $s_0 \in S$ is the initial state
3. $S_G \subseteq S$ is a non-empty set of goal states
4. $A(s)$ is the set of actions a applicable in state s, and
5. $next$ is a transition function that maps a state s into a successor state $s_a = next(a, s)$ for any action $a \in A(s)$

A *solution* of a state-state model is a finite sequence of applicable actions $a_0, a_1, .., a_n$ that maps the initial state s_0 into a goal state $s \in S_G$. That is, the action sequence a_0, a_1, .., a_n must generate a sequence of states s_i, $i = 0, \ldots, n+1$, such that $s_{i+1} = f(a_i, s_i)$, $a_i \in A(s_i)$, and $s_{n+1} \in S_G$. We say that two state spaces are *equivalent* when they have the same solutions. Often one is interested in solutions that are optimal in some sense; e.g., that minimize the number of actions. We'll say more about this in Sect. 7.

2.3 STRIPS STATE MODEL

A Strips planning problem $P = \langle \mathcal{L}_S, \mathcal{O}_S, \mathcal{I}_S, \mathcal{G}_S \rangle$ is given a precise meaning by mapping it into the state model $\mathcal{S}(P) = \langle S, s_0, S_G, A, next \rangle$ where

A1. the states s are sets of atoms from $\mathcal{L}_S$

A2. the initial state s_0 is $\mathcal{I}_S$

A3. the goal states are the states s such that $\mathcal{G}_S \subseteq s$

A4. $A(s)$ is the subset of operators $op \in \mathcal{O}_S$ such that $Prec(op) \subseteq s$

A5. the transition function $next$ is such that $next(a, s) = s + Add(a) - Del(a)$, for $a \in A(s)$

The solution of the planning problem P is the solution of the state model $\mathcal{S}(P)$; namely, a sequence of applicable actions that maps the initial state into a goal state in $\mathcal{S}(P)$.

For extensions of Strips such as those involving negated literals or conditional effects, slightly different state models are needed (Neb98). These different models, however, have something in common: they are all *propositional* in the sense that the internal structure of atoms can be ignored. In particular, different atoms can be substituted by different propositional symbols, resulting in state models that are equivalent. For Functional Strips, this is no longer the case as different terms may denote the same object. For constructing a state-model for Functional Strips, we will thus have to consider the internal structure of atoms, and hence will find convenient to distinguish two notions that are collapsed in the model [A1]–[A5]: the notion of *state* and the notion of *state representation.* We will associate the former with the *logical interpretations* over the state language (GL93; San94) and the latter with suitable encodings of them. For the basic Strips language, states are conveniently *represented* by a set of atoms, but for other languages, including Functional Strips, states are represented in a different way. To make the transition from Strips to Functional Strips simpler, we will thus first reformulate the state model [A1]–[A5] making this distinction explicit.

2.4 NON-PROPOSITIONAL STRIPS MODEL

An *interpretation* s is a mapping that assigns a denotation x^s to each symbol, term, and formula x in the language. In Strips, the symbols are constant or relational symbols. Constant symbols play the role of *object names,* with different names referring to different objects. We assume a finite set $\mathcal{C}$ of such constant symbols n and a finite domain of interpretation D such that different names denote different objects and all objects have names. Furthermore, we consider a class of interpretations in which the denotation of names, as opposed to the denotation of fluents, is *fixed.* That is, if we write n^* to refer to the denotation of names $n \in \mathcal{C}$, we consider only interpretations s for which $n^s = n^*$ for all $n \in \mathcal{C}$. We call the denotation function $* : \mathcal{C} \mapsto D$ that establishes a 1-to-1 correspondence between names and objects, the *representation function for constants.* The choice of this representation function is arbitrary in Strips as they all lead to state models that are equivalent (i.e., that have the same solutions).

The denotation p^s of relational symbols p of arity k is a subset of D^k. The denotation of relational symbols, unlike the denotation of constant symbols, can change as a result of the actions. Relational symbols are thus *fluent* symbols, while constant symbols are *fixed.* The distinction between fluent and fixed symbols is *semantic,* and one could think of languages where relational symbols are fixed and constant symbols are fluents. Languages with *fluent 'constant' symbols* such as $level\_of\_fuel$ are used in a number of planners (JB94; CT91; Wil88; Koe98; LG95; PW94) and action languages (San94; GKL97), where they are referred to as *state-variables, resources,* or *features*.

The denotation $[p(t)]^s$ of an atom $p(t)$, where p is a relational symbol in $\mathcal{P}$ and t is a tuple of terms of the same arity as p, is **true** if $t^s \in p^s$ and **false** otherwise. In Strips, the finite set of object names $n \in \mathcal{C}$ are the only terms, and hence the resulting set of atoms is finite.

The interpretations s that result from a given representation function for constants can be encoded by the set $[s]$ of atoms $p(t)$ that they make true. Taking into account the distinction between s and its representation $[s]$, the model [A1]–[A5] associated to a Strips problem $P = \langle \mathcal{L}_S, \mathcal{O}_S, \mathcal{I}_S, \mathcal{G}_S \rangle$ can be reformulated as follows:

B1. the states $s \in S$ are the *logical interpretations* over the language $\mathcal{L}_S$, and they are *represented* by the set $[s]$ of atoms that they make true

B2. the initial state s_0 is the interpretation that makes the atoms in $\mathcal{I}_S$ true and all other atoms false

B3. the goal states $s \in S_G$ are the interpretations that make the atoms in $\mathcal{G}_S$ true

B4. the actions $a \in A(s)$ are the operators $op \in \mathcal{O}_S$ whose preconditions are true in s

B5. the transition function $next$ maps states s into states $s' = next(a, s)$ for $a \in A(s)$ such that the representation of s' is $[s'] = [s] - Del(a) + Add(a)$.

It is simple to show that for any Strips problem the state models [A1]–[A5] and [B1]–[B5] are equivalent (they possess the same solutions). This equivalence is independent of the representation function for constants used. Model [B1]–[B5], however, is more flexible than model [A1]–[A5] as it can be easily modified to accommodate other languages. For example, *negation* can be accommodated by representing states by the *literals* they make true ([B1]) and by adjusting the representation of successor states ([B5]).[3] As we will see below, similar modifications are needed to accommodate *functions.*

[3] In addition, the initial state in [B2] has to be fully determined without closed world assumptions; see (Neb98).

3. FUNCTIONAL STRIPS

The language of Functional Strips differs from Strips in the two main aspects: function symbols are allowed and they can be fluents. These are small changes but with interesting consequences for modeling and problem solving.

3.1 MOTIVATION

As an illustration, let us consider the Towers of Hanoi problem. This is a standard problem in AI (NS72; Nil80; Pea83) that involves a number of disks of different sizes that have to be moved from one peg to another. Only disks at the top of a peg can be moved and they can never be placed on top of smaller disks.

The formulation of this problem in Strips requires relations like $on(i, j)$, $clear(i)$, and $smaller(i, j)$, and actions like $move(i, j, k)$ with i, j, and k ranging over the disk names.[4] If the number of disks is N, this implies in the order of N^3 ground actions. For $N = 10$, this means 1000 ground actions. These ground actions can be described very conveniently by means of schemas or quantification, yet most modern planners including Graphplan, SAT and heuristic planners (BF95; KS99; BG99) require the substitution of schemas by their ground instances. This causes a real computational problem if N is large. Interestingly, the number of actions that are ever *applicable* in a state is given by the number of *pegs* and not the number of *disks.* In particular, with 3 pegs, there can never be more than 3 ground actions applicable in any state. Still, even planners that rely on user-supplied control knowledge must scan the N^3 ground actions in order to identify the ones that are applicable.

The same problem appears in slightly different form in the extensions of Strips that accommodate conditional effects and quantification such as ADL (Ped89). In an ADL language, it's possible to formulate the problem so that the number of *ground actions* can be kept small but at the expense of a large number of *ground conditional effects.*[5] For example, using the relations $top(p_i, d_k)$ to state that the top disk in peg p_i is d_k, and $on(d_k, d_l)$ to express that disk d_k is on disk d_l, then the actions $move(p_i, p_j)$ can be defined over *pegs* so that when $top(p_i, d_k)$, $top(p_j, d_l)$, and $on(d_k, d_m)$ are all true, $top(p_i, d_m)$, $top(p_j, d_k)$, and $\neg top(p_i, d_k)$ be-

[4] This is a slight simplification as the bottom disks are not on top of other disks. This, however, is not relevant to our discussion.

[5] The recent ADL planners are based on Graphplan and also replace schemas by their ground instances; see (GK97; KNHD97; ASW98).

come true in the successor state. Yet again, there will be N^3 *ground conditional effects* associated with each action.

The problem with the large number of ground instances in Strips and ADL formulations is a consequence of the demand that objects be referred by their unique names. Indeed, once function symbols are allowed as 'first-class citizens' and compound terms can be used to name objects, this problem can be avoided.[6] Functional Strips is based on this idea and replaces the *relational* fluents in Strips with *functional* fluents. Provided with fluent *terms* like $top(p_i)$ and $top(p_j)$ representing the top disks in pegs p_i and p_j, the effects of the action $move(p_i, p_j)$ can be expressed as affecting the disks $top(p_i)$ and $top(p_j)$ directly, without having to appeal to the explicit names of those disks.

A complete formulation of Towers of Hanoi in Functional Strips is shown in Fig. 1.1. The most significant change from Strips is the use of postconditions of the form

$$f(t) := w$$

for terms $f(t)$ and w, in place of add and delete lists. A postcondition of that form says that in the state $s_a = next(a, s)$ that results from doing action a in state s, the denotation f^{s_a} of fluent f must become such that the equation

$$f^{s_a}(t^s) = w^s$$

holds, where t^s and w^s refer to the denotations of t and w in the state s.

The formulation in Fig. 1.1 uses the function $loc(d_k)$ to denote the disk below d_k and $size(d_k)$ to encode the size of disk d_k. A postcondition of the action $move(p_i, p_j)$ like $loc(top(p_i)) := top(p_j)$ therefore says that the action makes $loc(d_k) = d_l$ true when $d_k = top(p_i)$ and $d_l = top(p_j)$ are both true. This follows from the semantics sketched above and explained below in further detail. Note that the number of ground actions for the problem depends on the number of pegs but not on the number of disks.

3.2 LANGUAGE

The state language in Functional Strips (FStrips) is a first-order language with no quantification, involving *constant*, *function* and *relational symbols* but no variable symbols.

For simplicity, we assume that all *fluent* symbols are encoded as *function* symbols. Constant fluent symbols can be encoded as function sym-

[6] It must be noted that the original formulation of ADL accommodates function symbols but not as 'first-class citizens'; in particular, they cannot appear nested in the head of actions rules; see (Ped89).

Domains:	$Peg : p_1, p_2, p_3$; the pegs
	$Disk : d_1, d_2, d_3, d_4$; the disks
	$Disk* : Disk, d_0$; the disks and a dummy bottom disk 0
Fluents:	$top : Peg \mapsto Disk*$; denotes top disk in peg
	$loc : Disk \mapsto Disk*$; denotes disk below given disk
	$size : Disk* \mapsto Integer$; represents disk size
Action:	$\mathbf{move}(p_i, p_j : Peg)$; moves between pegs
Prec:	$top(p_i) \neq d_0$, $size(top(p_i)) < size(top(p_j))$
Post:	$top(p_i) := loc(top(p_i))$; $loc(top(p_i)) := top(p_j)$; $top(p_j) := top(p_i)$
Init:	$loc(d_1) = d_0$, $loc(d_2) = d_1$, $loc(d_3) = d_2$, $loc(d_4) = d_3$,
	$top(p_1) = d_4$, $top(p_2) = d_0$, $top(p_3) = d_0$,
	$size(d_0) = 4$, $size(d_1) = 3$, $size(d_2) = 2$, $size(d_3) = 1$, $size(d_4) = 0$
Goal:	$loc(d_1) = d_0$, $loc(d_2) = d_1$, $loc(d_3) = d_2$, $loc(d_4) = d_3$, $top(p_3) = d_4$

Figure 1.1 Formulation of `3-towers-of-hanoi` in Functional Strips

bols of arity 0, while relational fluent symbols can be encoded as function symbols of the same arity plus equality. This guarantees that any Strips representation can be easily translated into FStrips, even if FStrips often provides alternative encodings. For example, atoms in the blocks-world expressed as $on(a, b)$ can be encoded more conveniently in Functional Strips as $loc(a) = b$, where loc is a function symbol that makes explicit that blocks are at a single location.

As before, we call the non-fluent symbols in the language, the *fixed* symbols. They include all constant and relational symbols, as well as the function symbols that are not fluents. As in Strips, we assume that the denotation x^* of fixed symbols x is *fixed* by a *representation function* and consider only the interpretations s for which $x^s = x^*$. Among the fixed symbols we have a finite set $\mathcal{C}$ of *object names* that are assumed to refer to different objets, and constant, function, and relational symbols such as '3', '+', '=' that have a standard interpretation. The denotation of *fixed terms* t, i.e., terms involving no fluent symbols, does not depend on the state and is expressed as t^*. We call the terms that involve fluent symbols, *fluent terms.*

For the *representation* of states to be compact and finite, the language of Functional Strips is typed. The formulation of Hanoi, for example, involves the types Peg, $Disk$, and $Disk*$. The use of types is common in planning languages (McD98b) where they are used to delimit the range of action schemas. In FStrips, they also define the domains over which fluents are interpreted. For example the declaration $Disk : d_1, d_2, d_3, d_4$ says that the constant symbols d_i, $i = 1, 2, 3, 4$ denote objects d_i^* of type *Disk and* that there are no other *Disk* objects. Similarly, the declaration $Disk* : Disk, d_0$ says that the $Disk*$ objects are given by the *Disk* objects and the object denoted by d_0.

The arguments of functional fluents must range over domains that are *finite.* Examples of such domains are booleans, finite integer intervals, and enumerated domains like $Peg = \{p_1, p_2, p_3\}$. The representation function '$*$' for constants maps the standard symbols '=', '+', '3', etc, into their standard interpretation, and object names like p_1, p_2, etc, into different *integers* p_1^*, p_2^*, etc. Under suitable syntactic conditions, it can be proved that this mapping is irrelevant as long as different names are mapped into different objects.

3.3 OPERATORS

In FStrips, an operator *op* is described by the type of its arguments and two sets: the *precondition* and the *postcondition* lists, referred to as $Prec(op)$ and $Post(op)$. The precondition list is a set of *formulas,* while the postcondition list is a set of *updates* of the form:

$$f(t) := w \tag{1.1}$$

where $f(t)$ and w are terms of the same type, and f is a fluent symbol. Updates like (1.1) express how fluent f changes when an action is taken. Such postcondition says that the denotation f^{s_a} of f in the successor state $s_a = next(a, s)$ must become such that the following equation is satisfied:

$$f^{s_a}(t^s) = w^s \tag{1.2}$$

For example, an update like $h := h + 1$ means that h is incremented by 1, while an update like $loc(top(p_1)) := top(p_2)$ says that $loc(d_4) = d_3$ must become true in s_a when $top(p_1) = d_4$ and $top(p_2) = d_3$ are true in s.

Note that the terms t and w in (1.1) are interpreted in the state s in which the action a is taken, and affect the denotation of the fluent f in the next state. Postconditions, thus, do not interact. Also Equation 1.2 says nothing about the persistence of fluents; this is treated below.

3.4 FUNCTIONAL STRIPS STATE MODEL

The state-model for Functional Strips defines the semantics of the operators and the planning task. The model is similar to the non-propositional Strips model [B1]–[B5] where states are interpretations over the language and operators stand for updates on state representations. The differences arise from the differences in the language.

3.4.1 States and State Transitions. The states s are interpretations over the language $\mathcal{L}_F$ defined by the the fluents and constants. Given the representation function of constants, states s can be represented by the interpretation f^s of fluent symbols f only. We refer to the domain of the function f^s denoted by f as D_f. This domain is finite and is independent of the state s.[7]

Given a state s and an action a applicable in s, the successor state $s_a = next(a, s)$ is defined by the denotation f^{s_a} of each fluent symbol f in s_a. This denotation is given by the equation

$$f^{s_a}(v) = \begin{cases} w^s & \text{if } f(t) := w \text{ in } Post(a) \text{ and } v = t^s \\ f^s(v) & \text{otherwise} \end{cases} \tag{1.3}$$

where v ranges over the objects in D_f. This equation extends (1.2) with the standard assumption of fluent persistence (San94).

3.4.2 State Representation and Updates. Equation 1.3 provides the *declarative semantics* of actions in Functional Strips. The *operational semantics* is defined in terms of state *representations.*

A *state representation* in Functional Strips is a an assignment of *values* to a finite number of *state variables.* For each fluent f denoting functions with domain D_f and range R_f, we create a finite set of state variables $f[v]$, one for each $v \in D_f$. A state s is represented by an assignment in which each state variable $f[v]$ is assigned a value $f_s[v]$ in R_f that stands for $f^s(v)$. Thus, the denotation f^s of f is encoded by the value of the finite set of state variables $f[v]$ which are implemented as an array.

The representation of the state s_a that follows an action a in the state s is obtained from (1.3) as

$$f_{s_a}[v] = \begin{cases} w^s & \text{if } f(t) := w \text{ in } Post(a) \text{ and } v = t^s \\ f_s[v] & \text{otherwise} \end{cases} \tag{1.4}$$

This computation is linear in the number of effects as in Strips. The overhead comes from the evaluation of the terms t and w in the state s, but this is negligible in general.

3.4.3 State Variables and Terms. We say that a term $f(t)$, where f is a fluent symbol and t is a tuple of fixed terms, *refers* to a state variable $f[v]$, when t denotes v; i.e., $t^* = v$. When no confusion arises we also say that $f(t)$ *is* the state variable. For example, we say

[7] In general D_f is a *tuple* of domains of the same arity as f and the arguments taken by f are tuples as well. The presentation is simplified assuming that the arity of f is 1 but the generalization is straightforward.

that the state in the Hanoi problem is represented by the values of the state variables $top(p_1)$, ..., $top(p_3)$, $loc(d_0)$, ..., $loc(d_N)$, and $size(d_0)$, ..., $size(d_N)$.

A term $f(t)$ where t involves fluent symbols will normally refer to different state variables in different states. E.g., $loc(top(p_1))$ refers to the state variable $loc(d_1)$ in states where $top(p_1) = d_1$ and to $loc(d_2)$ in states where $top(p_1) = d_2$. It's precisely this treatment of fluent function symbols as 'first-class citizens' that distinguishes Functional Strips from other planning and action languages that accommodate state-variables such as (CT91; Wil88; JB94; LG95; PW94; Koe98; San94; GKL97).

3.4.4 Stating Problems. A planning problem in Functional Strips is a tuple $P = \langle \mathcal{L}_F, \mathcal{O}_F, \mathcal{I}_F, \mathcal{G}_F \rangle$ where $\mathcal{L}_F$ is the language, $\mathcal{O}_F$ the operators, and $\mathcal{I}_F$ and $\mathcal{G}_F$ are formulas standing for the initial and goal situations. The language $\mathcal{L}_F$ is defined by declaring the fluents and their domains, while operators are defined by means of suitable schemas. In addition, a representation function '*' that maps fixed symbols into their denotation is assumed. The representation function for standard symbols like '=', '+', 3, etc, is assumed to be provided by the underlying programming language, while the representation function for object names maps different names into different objects (integers).

The formulas $\mathcal{I}_F$ defining the initial situation must define a unique state that should be easy to compute. For this reason, we assume that the formulas in $\mathcal{I}_F$ must be of the special form $f(t) = w$, where f is fluent symbol and t and w are *fixed* terms (i.e., terms involving no fluent symbols). The initial state s_0 then is such that $f^{s_0}(t^*) = w^*$.[8]

3.4.5 State Model. All the ingredients are in place to define the state model associated with a problem $P = \langle \mathcal{L}_F, \mathcal{O}_F, \mathcal{I}_F, \mathcal{G}_F \rangle$ in Functional Strips. The state model is such that

- C1. the states $s \in S$ are the logical interpretations over the language $\mathcal{L}_F$, and are represented by assigning a value $f_s[v]$ to each state variable $f[v]$ for each fluent f and value v in D_f
- C2. the initial state s_0 satisfies the equations $f(t) = w$ in $\mathcal{I}_F$
- C3. the goal states $s \in S_G$ are the interpretations that satisfy the goal formula $\mathcal{G}_F$
- C4. the actions $a \in A(s)$ are the operators $op \in \mathcal{O}_F$ whose preconditions are true in s

[8] A partial characterization of the initial situation gives rise to a slightly different planning task that is usually referred to as *planning with incomplete information*. See (SW98; BG00).

C5. the representation of the next state $s_a = next(a, s)$ for $a \in A(s)$ is such that for each fluent symbol f and $v \in D_f$

$$f_{s_a}[v] = \begin{cases} w^s & \text{if } f(t) := w \text{ in } Post(a) \text{ and } v = t^s \\ f_s[v] & \text{otherwise (persistence)} \end{cases}$$

A number of examples will be used to illustrate the language.

4. EXAMPLES

4.1 BLOCKS

The blocks world domain is a convenient testbed for modeling and planning. The most common encoding in Strips involves an action $move(x, y, z)$ for moving a block x from a block y onto a block z, as well as actions for moving blocks to the table and from the table. In the presence of N blocks, this encoding leads to three action schemas with more than N^3 ground instances. As with Towers of Hanoi, ADL can model the problem with a single conditional schema, but the number of ground conditional effects remains N^3. An alternative often used in planning is to decompose the actions $move(x, y, z)$ into two actions with two arguments each: $unstack(x, y)$ and $stack(y, z)$. Similar actions are defined for placing and removing blocks from the table. Such decomposition reduces the number of ground actions to N^2 but makes planning harder by adding more choice points in the search.

In Functional Strips, it is possible to model this problem with N^2 ground actions and a single schema (Fig. 1.2). The actions $move(x, y)$ moves a block x to a location y that can be another block or the table. The block on which x was located, denoted by $loc(x)$, becomes *clear* after the action, but its explicit name is not needed. We use *Bool* as the domain given by the boolean terms **true** and **false**, and assume functions analogous to '=', '¬', etc. that return boolean terms. Such functions can be easily defined in the underlying programming language. The postcondition $clear(y) := (y = table)$, for example, says that y becomes not *clear* if y is not the table. We also abbreviate terms of the form $t = \textbf{true}$ and $t = \textbf{false}$ by t and $\neg t$ respectively.

4.2 LOGISTICS

Logistics is a more recent benchmark in planning that deals with the transportations of packages (Vel92; KS96; McD98a). Packages are transported in trucks among locations in the same city (including airports) and by planes among airports in different cities. In the Strips formulation one needs schemas for actions like

Domains :	$Block : a, b, c, \ldots$; the blocks
	$Loc : Block, table$; the locations: blocks + table
Fluents:	$loc : Block \mapsto Loc$
	$clear : Loc \mapsto Bool$
Action:	$\mathbf{move}(x : Block, y : Loc)$
Prec:	$clear(x)\,;\ clear(y)\,;\ x \neq y$
Post:	$loc(x) := y\,; clear(y) := (y = table)\,; clear(loc(x)) := \mathbf{true}$
Init:	$loc(a) = table\,; loc(b) = a\,; \ldots, clear(b), clear(table)$
Goal:	$loc(a) = b,\ loc(b) = table$

Figure 1.2 Formulation of `Blocks-world` in Functional Strips

$$\mathbf{load}(pkg, transpt, loc)$$
$$\mathbf{unload}(pkg, transpt, loc)$$
$$\mathbf{drive\_truck}(truck, loc1, loc2)$$
$$\mathbf{fly\_plane}(plane, loc1, loc2)$$

where *transpt* refers to trucks and planes. In Functional Strips, the presence of functional fluents allows for a more concise representation in which the number of action arguments can be reduced (e.g., **unload** requires the package argument only). This encoding is shown in Fig. 1.3.

The function *city* is defined as a fluent even though it is fixed in the initial situation and doesn't change. A straightforward extension would allow us to declare such symbols as parameters rather than fluents (actually such extension is supported in the PDDL standard (McD98b)). We also make use of type predicates like $airport?(t)$ to test if t denotes an object of type *Airport*.

4.3 8-PUZZLE

The 8-puzzle is a standard problem in heuristic search (Nil80; Pea83). In Functional Strips, the description of the problem is very compact due to the ability to nest fluents and attach user defined functions to symbols (Fig. 1.4). There are two main *fluent* symbols: $tile : Pos \mapsto Tile$ that maps each grid position to the tile that occupies the position, and *bp* that keeps track of the position of the 'blank' tile. In addition, there are *four* symbols, u, l, ... that represent functions that map a position into each one of its four neighboring position (positions outside the grid are denoted by 0). For example, assuming that the top row positions are 1, 2 and 3, the second row positions are 4, 5, and 6, and so on, we must have $u(5) = 2$, $u(2) = 0$, etc. This interpretation of the symbols u, l, ... can be defined *extensionally* by modeling them as fluents and enumerating these equations in the initial situation, or *intensionally* by

Domains:	$Pkg : o_1, \ldots, o_{10}$
	$Truck : t_1, t_2, \ldots, t_4$
	$Plane : p_1, p_2, \ldots, p_3$
	$City : bos,\ pgh,\ lax,\ \ldots$
	$Airpt : abos,\ pgh,\ alax,\ \ldots$
	$Loctn : bos_1,\ bos_2,\ pgh_1,\ lax_1,\ \ldots$
	$Site : Airpt, Loctn\ ;\ Transp : Truck, Plane$
	$Thing : Transp, Pkg\ ;\ Loc : Transp, Site$
Fluents:	$loc : Thing \mapsto Loc$
	$city : Site \mapsto City$
Action:	$\mathbf{load}(pkg : Pkg, target : Transp)$
Prec:	$loc(pkg) = loc(target)$
Post:	$loc(pkg) := target$
Action:	$\mathbf{unload}(pkg : Pkg)$
Prec:	$transp?(loc(pkg))$
Post:	$loc(pkg) := loc(loc(pkg))$
Action:	$\mathbf{drive\_truck}(t : Truck, dest : Site)$
Prec:	$city(loc(t)) = city(dest)$
Post:	$loc(t) := dest$
Action:	$\mathbf{fly\_plane}(p : Plane, dest : Airprt)$
Prec:	$airport?(loc(p))$
Post:	$loc(p) := dest$
Init:	$city(abos) = bos,\ city(bos_1) = bos,\ \ldots,$
	$loc(t_1) = abos,\ loc(t_2) = apgh,\ loc(t_3) = lax_1,$
	$loc(p_1) = abos,\ loc(p_2) = alax,\ \ldots$
	$loc(o_1) = bos_1,\ loc(o_2) = bos_2,\ loc(o_3) = pgh_1$
Goal:	$loc(o_1) = pgh,\ loc(o_2) = pgh,\ loc(o_3) = alax,\ \ldots$

Figure 1.3 Formulation of `Logistics` in Functional Strips

Domain:	Pos : $1, \ldots, 9$ Tile : $0, \ldots, 8$ Pos* : Pos, 0
Fluent:	$tile : Pos \mapsto Tile$ $bp : Pos$
Fixed:	$u, l, r, d : Pos \mapsto Pos*$
Action:	**up**
Prec:	$u(bp) \neq 0$
Post:	$bp := u(bp)$; $tile(bp) := tile(u(bp))$; $tile(u(bp)) := tile(bp)$
Action:	**down**, **left**, **right**, ...
Init:	$tile(1) = 2$, $tile(2) = 0$, $tile(3) = 2$, ..., $tile(9) = 5$, $bp = 2$
Goal:	$tile(1) = 1$, $tile(2) = 2$, $tile(3) = 3$, ..., $tile(9) = 8$

Figure 1.4 Formulation of `8-puzzle` in Functional Strips

modeling these symbols as fixed symbols with an interpretation provided in the underlying language. In such case, a function u^* must be defined in the underlying language such that $u^*(x)$ returns 0 if $x \leq 3$ and returns $x - 3$ otherwise. Something similar has to be done for the other functions d^*, r^*, and l^*. The declaration that u, d, r, and l are **fixed** symbols in Fig. 1.4 indicates that their denotation is defined in the underlying programming language.

In the resulting model, the actions **up**, **left**, ..., have no arguments. Indeed, the number of ground actions matches exactly the branching factor of the problem. In addition, the resulting state representation is in close correspondence with the representation used in specialized programs. It's worth emphasizing that these are features that are necessary (although not sufficient) for making planning approaches competitive with specialized solvers.

5. RESOURCES

The word 'resources' in planning and scheduling refers to objects that can be produced, consumed, or 'borrowed' during the execution of plans, constraining the possible actions (Wil88; CT91; LG95; PW94; EKR96; Koe98). E.g., driving a car requires and consumes fuel, building a wall requires and consumes bricks, etc. An implicit assumption when a collection of objects is represented as a resource is that the identity of the objects in the set does not matter. This is important as actions and states that would be different if objects were named individually are collapsed. Namely, actions like 'grabbing brick1', 'grabbing brick2', and so on, are replaced by the single action 'grabbing a brick'. Such simplified representations can have a significant impact on planning (Wil88; CT91).

Resources are usually represented as real or integer state variables. In certain cases, such state variables can depend on one or more arguments (e.g., $level\_fuel(car_1)$). Functional Strips, by making function symbols 'first-class citizens', allows these and other uses of state-variables. In particular, state-variables can be nested, allowing for terms like $B(loc)$ where both symbols loc and B are fluents. We illustrate the uses of such constructs by considering a variation of the 'Gripper' domain used in the AIPS-98 Planning Competition (McD98a).

'Gripper' involves a robot with a number of grippers that can move between rooms, picking up and dropping balls from its grippers (one ball per gripper at most). In the competition, this problem proved to be hard, and three out of the four competing planners solved a few instances only. The Strips formulation of Gripper involves names for each ball and each gripper, along with predicates for keeping track of the status of each one of them (the location of balls, whether a gripper is free or not, etc). As discussed in (FL99), this representation produces a number of symmetries that if exploited at run-time can improve planner performance significantly. Alternatively, these symmetries can often be exploited at *modeling time.* For example, if we do not care about the identity of the individual balls and grippers, balls and grippers can be modeled as *resources.* This leads to substantial simplifications in both the branching factor of the problem and the state representation. The resource formulation of Gripper in Functional Strips is shown in Fig. 1.5. While in the Strips formulation, the actions $move$, $pick$ and $drop$ involve 2 or 3 arguments, in the 'resource' formulation only the action $move$ needs an argument. The fluent loc keeps track of the room where the robot is, B keeps track of the number of balls in each room, and G stands for the number of grippers. A term like $B(loc)$ thus denotes the number of balls in the room where the robot is located. The denotation of this term changes when either the number of balls in the room or the position of the robot changes. The state representation that results from the formulation in Fig. 1.5 is once again as economical as the representation that can be obtained in a specialized program.

6. PLANNING AND PROBLEM SOLVING

In this section we discuss briefly how Functional Strips can be used in planning and problem solving. We distinguish two cases. In *domain-independent planning,* the domain descriptions are assumed to encode all the knowledge needed to solve the problem, including the dynamics of the domain and the control knowledge. In *specialized problem solving,* on the other hand, domain descriptions encode the dynamics of the domain

Domain:	$Room : a, b, c, \ldots$; the rooms
Fluents:	$loc : Room$; the room where the robot is
	$G : Int$; number of grippers
	$B : Room \mapsto Int$; number of balls in each room
	$H : Int$; number of balls being held
Action:	**move**$(dest : Room)$
Prec:	–
Post:	$loc := dest$
Action:	**pick**
Prec:	$H < G$; $B(loc) > 0$
Post:	$H := H + 1$; $B(loc) := B(loc) - 1$
Action:	**drop**
Prec:	$H > 0$
Post:	$H := 0$; $B(loc) := B(loc) + H$
Init:	$G = 2$, $H = 0$, $loc = a$, $B(a) = 20$, $B(b) = 0$
Goal:	$B(b) = 15$

Figure 1.5 Formulation of `Gripper` in Functional Strips

but the control knowledge can be provided separately (e.g., in the form of an heuristic function or a set of control rules). We focus first on the latter case.

6.1 PROBLEM SOLVING WITH FUNCTIONAL STRIPS

Consider writing a program for solving a problem like the Rubik's cube. One option is to write the program in a programming language such as C. One would then write some routines for modeling the dynamics of the problem and other routines for capturing the control; namely, which action to try next, when and where to backtrack, etc. This is actually the most common option for solving combinatorial problems and it's the approach taken for example in (Kor98). The advantage of this approach is that it can be very efficient at *run time;* the disadvantage, is that it may be quite inefficient at *modeling time.* That is, building a good specialized program takes time, and usually involves a tedious process of debugging and tuning.

A modeling language such as Functional Strips can be used in this setting to reduce *modeling time* without incurring in a substantial overhead at *run time.* For that, Functional Strips can be used for describing the dynamics of the domain which can then be automatically compiled into efficient run-time procedures (i.e., procedures for testing when an action is applicable in a state and for computing successor states). These compiled procedures can take the place of the routines written by hand. They will impose a minimal overhead if the encoding of the problem is such that the resulting state representation is in correspondence with the state representation used by a specialized program. As argued above, this can often be achieved in Functional Strips but is more difficult to achieve in Strips or ADL languages where the number of ground actions often explodes and state representations have often little to do with specialized representations.

We have actually implemented a tool that accepts descriptions of problems in Functional Strips and compiles them into state space search procedures. The heuristic function used by these procedures is supplied by the user in the form of a C++ routine. This tool can be seen as a domain-*dependent* planner in the style of TLPLAN (BK98) but while in TLPLAN the control knowledge is expressed in a logical language, in this tool, the control knowledge is expressed in the form of an heuristic function. Recent heuristic search planners like HSP are based on a similar idea but rather than relying on a user-supplied heuristic function, they extract the heuristic function automatically from the problem description. We elaborate on this below.

6.2 PLANNING WITH FUNCTIONAL STRIPS

HSP is a planner that maps Strips problems into state-space search problems that are solved with an heuristic extracted from the Strips encodings (BG99). For a state s, the heuristic value $h(s)$ that estimates the distance from s to the goal is obtained by computing the cost of achieving each atom p from s under some simplifying assumptions. The value $h(s)$ is then set to the sum of the costs of the atoms p in the goal. While the heuristic values $h(s)$ are not admissible (they may overestimate the true cost to the goal), they can be computed reasonably fast and are often quite informative (they drive the search in a good direction). From the results in the recent AIPS Planning Competition (McD98a), HSP appears competitive with the state of the art Graphplan and SAT planners (LF99; KNHD97; KS99).

The ideas of HSP can be used in the context of Functional Strips (see (BLG97) for related results). The key point is the automatic extraction of the heuristic from Functional Strips encodings. We have been exploring a number of ways for doing this, but coming up with an efficient implementation that can be competitive with HSP on similar problems has been difficult. The problem is that the effect of postconditions $f(t) := w$ in Functional Strips is *state dependent* when the terms t or w involve fluent symbols. An alternative that we are currently exploring for computing the heuristic is to translate this context-dependent updates into equivalent sets of *conditional* but *context-independent* updates of the form $C \rightarrow f(t') := w'$, where t' and w' have no fluent symbols and $C = (t = t') \wedge (w = w')$.

Approaches that do not involve the extraction of an heuristic function are also possible. For example, (BC99) maps planning problems into *constraint satisfaction problems* that are then solved by domain-independent methods. This approach may prove suitable for Functional Strips where the state is represented by a finite set of state variables taking a finite set of values. However, more empirical work is needed to evaluate how the different planning approaches can benefit from the additional facilities in the modeling language.

7. DISCUSSION

The work we have presented is motivated by a perspective in which *planning is general problem solving.* As such, planning should offer a general language for expressing problems and general algorithms for solving them. For such an approach to be useful though, the time required to model problems and find the solutions has to be competitive with the time required to set up and solve specialized models. The quality of the solutions has to be competitive as well. This will be possible only by a suitable combination of general and effective languages and algorithms as shown by the closely related work in the area of Constraint Programming (HSD92; MS99).

Due to the similarity between Functional Strips, Strips, and the $\mathcal{A}$ action representation language (GL93), the extensions found in ADL (conditional effects, negation, and quantification (Ped89)) and languages such as $\mathcal{AR}$ (ramifications) (GKL97) can easily be integrated with the extensions accommodated in Functional Strips. Among the other extensions that are likely to be necessary in a good modeling language for planning, we mention the following:

- **Constraints.** Constraints represented as formulas can be used to provide implicit action preconditions (e.g., (GKL97)). Namely, the set $A(s)$ of actions applicable in s will exclude all actions a that lead to states $s_a = next(a, s)$ that violate a constraint. Such constraints can be used to express capacity constraints (e.g., that the number of balls being held cannot exceed the number of grippers) or control knowledge (BK98). With constraints, any Constraint Satisfaction Problem (CSP) can be expressed as a Planning problem. The question is how to make those constraints play an active role in the search. So far, only approaches based *exclusively* on constraint techniques are able to do that (e.g., (KS99; BC99)). A combination of constraint-directed and heuristic search techniques may also prove useful.

- **Costs.** The state models discussed above can be extended to take into account action costs. Traditionally, the focus in planning has been on *uniform* action costs, however, costs may depend on both actions and states. For example, a *fluent* t standing for *time* can be included so that actions increase t by their time duration. Then, defining $c(a, s)$ as the duration of a in s, i.e., $c(a, s) = (t^{s_a} - t^s)$ where $s_a = next(a, s)$, a formulation would be obtained in which optimal plans stand for plans with minimum completion times. In a different way, optimal *parallel plans,* such as those computed by Graphplan and SAT planners, can be defined as well. The issue is how to plan effectively in such flexible cost structures.

- **Data Structures.** The state representation of the 8-puzzle is close to the representation one could find in a specialized program. However, the same cannot be said for Towers of Hanoi. In a specialized program, the state would not be represented by a set of equalities $loc(disk_i) = disk_j$, but by three lists $(disk_1, disk_2, \ldots)$ one for each peg, meaning that $disk_1$ is on $disk_2$ which is on $disk_3$, etc. Then the action of moving a disk from one peg to another would be implemented as operations on the heads of these lists. One way to achieve such efficient representations in FStrips is by allowing Lists as primitive types. Then functions like *car*, *cdr*, *cons*, etc. defined in the underlying language can be made available in the planning language. This would allow high-level representations of Tower of Hanoi and other problems that are as efficient as specialized representations.

We have developed a tool that supports some of these extensions. The tool also accommodates probabilistic actions and partial observability; see (BG98; BG00).

Acknowledgements: The tool discussed in the paper has been built by Blai Bonet. Blai has also provided valuable feedback on a number of topics related to this paper. Part of this work was done while I was visiting IRIT, Toulouse, and Linköping University. I thank J. Lang, D. Dubois, H. Prade, and H. Farreny in Toulouse, and E. Sandewall and P. Doherty in Linköping for their hospitality and for making these visits possible. I also thank Chitta Baral for useful comments.

References

C. Anderson, D. Smith, and D. Weld. Conditional effects in graphplan. In *Proc. AIPS-98*. AAAI Press, 1998.

P. Van Beek and X. Chen. CPlan: a constraint programming approach to planning. In *Proc. AAAI-99*, 1999.

A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, Montreal, Canada, 1995.

B. Bonet and H. Geffner. High-level planning and control with incomplete information using POMDPs. In *Proceedings AAAI Fall Symp. on Cognitive Robotics*, 1998.

B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*. Springer, 1999.

B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AI Planning Systems*. AAAI Press, 2000. To appear.

F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning, 1998. Submitted.

B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719. MIT Press, 1997.

K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

A. El-Kholy and B. Richards. Temporal and resource reasoning in planning: the parcPLAN approach. In W. Wahlster, editor, *Proc. ECAI-96*, pages 614–618. Wiley, 1996.

M. Fox and D. Long. The detection and exploitation of symmetry in planning domains. In *Proc. IJCAI-99*, 1999.

R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

B. Gazen and C. Knoblock. Combining the expressiveness of ucpop with the efficiency of graphplan. In *Proc. 4th European Conf. on Planning*, volume LNAI 1248. Springer, 1997.

E. Giunchiglia, N. Kartha, and V. Lifschitz. Representing action: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *J. of Logic Programming*, 17:301–322, 1993.

P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1–3):113–159, 1992.

P. Jonsson and C. Bäckstrom. Tractable planning with state variables by exploiting structural restrictions. In *Proc. AAAI-94*, 1994.

R. Khardon. Learning action strategies for planning domains. Technical Report TR-09-97, Harvard, 1997. To appear in AI Journal.

J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. 4th European Conf. on Planning*, volume LNAI 1248. Springer, 1997.

J. Koehler. Planning under resource constraints. In *Proc. ECAI-98*. Wiley, 1998.

R. Korf. Finding optimal solutions to to Rubik's cube using pattern databases. In *Proceedings of AAAI-98*, pages 1202–1207, 1998.

H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201, 1996.

H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. In *Proceedings IJCAI-99*, 1999.

D. Long and M. Fox. The efficient implementation of the plan-graph. *JAIR*, 10:85–115, 1999.

P. Laborie and M. Ghallab. Planning with sharable resources constraints. In *Proc. IJCAI-95*, pages 1643–1649. Morgan Kaufmann, 1995.

V. Lifschitz. On the semantics of STRIPS. In *Proc. Reasoning about Actions and Plans*, pages 1–9. Morgan Kaufmann, 1986.

D. McDermott. AIPS-98 Planning Competition Results. `http://ftp.-cs.yale.edu/pub/mcdermott/aipscomp-results.html`, 1998.

D. McDermott. PDDL – the planning domain definition language. Available at `http://ftp.cs.yale.edu/pub/mcdermott`, 1998.

K. Marriot and P. Stuckey. *Programming with Constraints*. MIT Press, 1999.

B. Nebel. On the compilability and expressive power of propositional planning formalisms. Technical report, Freiburg University., 1998. At `http://www.informatik.uni-freiburg.de/ nebel`.

N. Nilsson. *Principles of Artificial Intelligence*. Tioga, 1980.

A. Newell and H. Simon. *Human Problem Solving.* Prentice–Hall, Englewood Cliffs, NJ, 1972.

J. Pearl. *Heuristics.* Morgan Kaufmann, 1983.

E. Pednault. ADL: Exploring the middle ground between Strips and the situation calcules. In *Proc. KR-89*, pages 324–332, 1989.

J. Penberthy and D. Weld. Temporal planning with continous change. In *Proc. AAAI-94*, pages 1010–1015, 1994.

R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

E. Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamical Systems.* Oxford Univ. Press, 1994.

M. Shanahan. *Solving the Frame Problem.* MIT Press, 1997.

D. Smith and D. Weld. Conformant graphplan. In *Proceedings AAAI-98*, pages 889–896. AAAI Press, 1998.

M. Veloso. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Computer Science Department, CMU, 1992. Tech. Report CMU-CS-92-174.

D. Wilkins. *Practical Planning: Extending the classical AI paradigm.* M. Kaufmann, 1988.