

# Planning with Preferences Using Constraint Logic Programming

Phan Huy Tu, Tran Cao Son, and Enrico Pontelli

Computer Science Department,  
New Mexico State University, Las Cruces, New Mexico, USA  
tphan|tson|epontelli@cs.nmsu.edu

**Abstract.** We describe the development of a constraint logic programming based system, called CPP, which is capable of generating most preferred plans with respect to a user’s preference and evaluate its performance.

## 1 Introduction

The problem of finding a plan satisfying the preferences of a user has been discussed in [13] and has recently attracted the attention of researchers in planning [3, 5, 2]. Indeed, PDDL, the de-facto language of the planning community, has been extended [9] to include constructs for the specification of users’ preferences. As one of the first efforts to integrate users’ preferences into a planning system, [13] proposed an answer set programming based system for computing a most preferred plan. Due to the lack of a list operator and the inflexibility in dealing with function symbols, the encoding of preferences in this system is not natural and requires the introduction of artificial constants and predicates symbols. For example, the formula  $(p \wedge q) \vee r$  is translated into the set of atoms  $\{or(n_1, n_2, r), and(n_2, p, q)\}$  with two new constants,  $n_1$  and  $n_2$ , where  $n_1$  is the “name” assigned to the original formula, and  $n_2$  is the “name” assigned to the formula  $(p \wedge q)$ . Furthermore, the encoding proposed in [13] also suffers from the requirement of answer set solvers that all variables have to be instantiated before computing the answer sets. Let us consider the rule

$$w(n_\psi, S) \leftarrow w(n_{\psi_1}, S_1), \dots, w(n_{\psi_k}, S_k), S = \sum_{r=1}^k 2^{k-r} \times S_r$$

which is used for computing the weight function of an atomic preference  $\psi = \psi_1 \triangleleft \psi_2 \triangleleft \dots \triangleleft \psi_k$  where  $n_\psi$  and  $n_{\psi_i}$ ’s are preference names of  $\psi$ ,  $\psi_i$ ’s respectively, and  $S$  and  $S_i$ ’s are positive numbers. Under the assumption that variables  $S$  and  $S_i$ ’s can take values from 0 to  $n$ , this rule is instantiated to  $(n + 1)^{k+1}$  distinct rules. For these reasons, the answer set programming based encoding in [13] would not scale well to handle complex preferences.

This paper proposes an alternative implementation scheme to address these problems, by adopting a different logic programming paradigm, i.e., Constraint Logic Programming (CLP) [11]. The choice of CLP is suggested by a number of factors. First of all, CLP is still a logic programming based paradigm, it is very declarative, and it allows us to reuse parts of the problem encoding we studied in [13]. On the other hand, CLP does not require program grounding and it allows

the use of lists and other function symbols, leading to a more compact encoding of parts of the problem. CLP (and, in particular, CLP over finite domains, used in this study [14]) provides the ability to express and efficiently handle arithmetic constraints, and the paradigm offers methodologies for describing search strategies and expressing optimization problems. These last two features appear to be vital in the context of dealing with preferences.

Preliminary studies have been conducted to compare answer set programming and CLP in encoding traditional planning problems (without preferences) [7]. It indicates that, while answer set programming offers more compact and declarative encodings in many problems, CLP provides for significantly more efficient and scalable executions.

In this paper, we present our development of a constraint logic programming based system, called CPP, for computing most preferred plans of deterministic and complete action theories. The language  $\mathcal{AL}$  [1] is chosen for representing action theories. To specify preferences we use the language  $\mathcal{PP}$  from [13]. The rest of this paper is organized as follows. Section 2 briefly summarizes the syntax and semantics of (a variant of)  $\mathcal{AL}$  and  $\mathcal{PP}$ . Section 3 presents our formulation of planning with preferences as a constraint satisfaction problem. Section 4 describes the implementation of CPP. Section 5 shows some experimental results with CPP. Finally, we conclude the paper and discuss future work in Section 6.

## 2 Preliminaries

### 2.1 The Action Language $\mathcal{AL}$

The alphabet of an action theory in  $\mathcal{AL}$  consists of a set of action names  $\mathbf{A}$  and a set of fluent names  $\mathbf{F}$ . A (fluent) literal is either a fluent  $f \in \mathbf{F}$  or its negation  $\neg f$ . A (fluent) formula  $\varphi$  is constructed from literals and connectives  $\wedge, \vee, \neg$  as usual. To describe an action theory, propositions of the following forms are used:

$$\mathbf{initially}(l) \tag{1}$$

$$\mathbf{causes}(a, l, p) \tag{2}$$

$$\mathbf{caused}(p, l) \tag{3}$$

$$\mathbf{executable}(a, p) \tag{4}$$

where  $a$  is an action,  $l$  is a literal, and  $p$  is a set of literals. Proposition (1) says that  $l$  holds in the initial state. Proposition (2), called a *dynamic law*, says that if  $a$  is performed in a state wherein  $p$  holds then  $l$  holds in the successor state. (3), called a *static law*, says that in any state in which  $p$  holds, then so does  $l$ . (4) is called an executability law which states that  $a$  is executable whenever  $p$  holds.

An *action theory* is given by a pair  $(D, I)$  where  $D$  is a set of propositions (2)–(4) and  $I$  is a set of propositions (1).  $D$  and  $I$  are called the *domain description* and *initial state condition*, respectively. A *planning problem* is a 3-tuple  $(D, I, G)$ , where  $(D, I)$  is an action theory and  $G$  is a set of literals. The semantics of an action theory  $(D, I)$  is given by the notion of *state* and by a transition function  $\Phi$  that specifies the result of the execution of an action  $a$  in a state  $s$  as follows.

Let  $l$  be a literal and  $\sigma$  be a set of literals.  $l$  holds in  $\sigma$ , denoted by  $\sigma \models l$ , if  $l \in \sigma$ . A set of literals  $\gamma$  holds in  $\sigma$ , denoted by  $\sigma \models \gamma$ , if  $\gamma \subseteq \sigma$ .  $\sigma$  is consistent if it does not contain two contrary literals.  $\sigma$  is complete if for each  $f \in \mathbf{F}$  either  $f$  or  $\neg f$  belongs to  $\sigma$ .  $\sigma$  satisfies a static law (3) if whenever  $\sigma \models p$ , we have  $\sigma \models l$ . We denote by  $Cl(\sigma)$  the smallest set of literals that includes  $\sigma$  and satisfies all the static laws in  $D$ . A *state*  $s$  of  $(D, I)$  is a complete and consistent set of literals that satisfies all the static laws in  $D$ . The truth value of a formula  $\varphi$  in  $s$  is defined as usual. When  $\varphi$  is true (resp. false) in  $s$  we write  $s \models \varphi$  (resp.  $s \not\models \varphi$ ). An action  $a$  is *executable* in  $s$  if there exists an executability condition (4) s.t.  $s \models p$ .

For a state  $s$  and an action  $a$  executable in  $s$ , the direct effects of  $a$  are defined as  $e(a, s) = \{l \mid \mathbf{causes}(a, l, p) \in D, s \models p\}$ . A state  $s'$  is a *possible successor state* of  $s$  after performing  $a$  iff  $s' = Cl(e(a, s) \cup (s \cap s'))$ . We denote by  $\Phi(a, s)$  the set of possible successor states of  $s$  after the execution of  $a$ . A state  $s_0$  is called a *possible initial state* of  $(D, I)$  if for every proposition **initially**( $l$ ) in  $I$  we have  $l \in s_0$ .  $(D, I)$  is *complete* if exactly one possible initial state exists.  $(D, I)$  is *consistent* if for any state  $s$  and action  $a$  executable in  $s$ ,  $\Phi(a, s) \neq \emptyset$ .  $(D, I)$  is *deterministic* if for every action  $a$  and state  $s$  such that  $a$  is executable in  $s$ , we have  $|\Phi(a, s)| \leq 1$ ; otherwise,  $(D, I)$  is non-deterministic. In this paper, we only consider consistent, complete, and deterministic action theories.

A *trajectory*  $\alpha$  is an alternate sequence of states and actions  $s_0 a_0 \dots s_{n-1} a_{n-1} s_n$  such that for  $1 \leq i \leq n$ ,  $a_{i-1}$  is executable in  $s_{i-1}$  and  $s_i \in \Phi(a_{i-1}, s_{i-1})$ . We call  $s_0$  and  $s_n$  the initial state and final state of  $\alpha$  respectively.  $\alpha$  is said to achieve a set of literals  $\sigma$  if  $\sigma$  holds in the final state of  $\alpha$ . For convenience, we write  $\alpha[i]$  ( $0 \leq i \leq n$ ) to denote the trajectory  $s_i a_i s_{i+1} a_{i+1} \dots a_{n-1} s_n$ .

Given a planning problem  $\mathcal{P} = (D, I, G)$ , a sequence of actions  $\pi = a_0 a_1 \dots a_{n-1}$  is a *plan* to  $\mathcal{P}$  if there exists a trajectory  $\alpha = s_0 a_0 s_1 \dots s_{n-1} a_{n-1} s_n$  such that  $s_0$  is the initial state<sup>1</sup> of  $(D, I)$  and  $\alpha$  achieves  $G$ . Observe that because we are assuming that  $(D, I)$  is complete and deterministic, if such  $\alpha$  exists then it is unique. Therefore, for a plan  $\pi$ , we call such  $\alpha$  *the trajectory* of  $\pi$  and denote it by  $\pi^*$ .

## 2.2 The Preference Language $\mathcal{PP}$

The language  $\mathcal{PP}$  allows for three types of preferences: *basic desires*, *atomic preferences*, and *general preferences*. Most preferred plans with respect to a preference  $\psi$  are defined based on the ordering relation  $\prec$  between trajectories of plans. Specifically, a plan  $\pi$  is *most preferred* w.r.t.  $\psi$  if there exists no plan  $\pi'$  such that the trajectory of  $\pi'$  is preferred to the trajectory of  $\pi$ , i.e.,  $\pi'^* \prec \pi^*$ .

In this section, we review these three types of preferences and how the relation  $\prec$  between trajectories w.r.t. each type of preferences is defined.

**Basic desires.** Two primitive forms of basic desires are *state desires* and *goal preferences*. A state desire is either a fluent formula  $\varphi$  or a formula  $occ(a)$  where  $a$  is an action. Intuitively, a state desire describes a basic user preference to be considered in the context of the “current” state of a trajectory. In many cases, it is also helpful to have the final state satisfy a fluent formula  $\varphi$  — this is called a goal preference and denoted by **goal**( $\varphi$ ). A (general) basic desire is defined as follows.

<sup>1</sup> Since we are assuming that  $\mathcal{P}$  is complete, such an initial state uniquely exists

**Definition 1.** A basic desire is either a state desire, a goal preference or a formula of the form  $\psi_1 \wedge \psi_2$ ,  $\psi_1 \vee \psi_2$ ,  $\neg\psi_1$ , **next**( $\psi_1$ ), **until**( $\psi_1, \psi_2$ ), **always**( $\psi_1$ ), or **eventually**( $\psi$ ) where  $\psi_1$  and  $\psi_2$  are basic desires.

For example, in the travel domain [13], to express the fact that a user prefers to go by taxi or bus from home to school, we can write **eventually**( $\text{occ}(\text{bus}(\text{home}, \text{school})) \vee \text{occ}(\text{taxi}(\text{home}, \text{school}))$ ). If the user's desire is not to call a taxi, we can write **always**( $\neg \text{occ}(\text{call\_taxi}(\text{home}))$ ). If for some reasons, the user's desire is not to see any taxi around his home, the desire **always**( $\neg \text{available\_taxi}(\text{home})$ ) can be used.

Having defined basic desires, we now define what it means by a trajectory satisfying a basic desire.

**Definition 2.** Let  $\alpha = s_0 a_0 s_1 a_1 s_2 \cdots a_{n-1} s_n$  be a trajectory, and let  $\psi$  be a basic desire. We say that  $\alpha$  satisfies  $\psi$  and write  $\alpha \models \psi$  iff one of the following holds

- $\psi = \text{goal}(\varphi)$  and  $s_n \models \varphi$
- $\psi = \varphi$  and  $s_0 \models \varphi$
- $\psi = \text{occ}(a)$ ,  $a_0 = a$ , and  $n \geq 0$
- $\psi = \psi_1 \wedge \psi_2$ ,  $\alpha \models \psi_1$  and  $\alpha \models \psi_2$
- $\psi = \psi_1 \vee \psi_2$ ,  $\alpha \models \psi_1$  or  $\alpha \models \psi_2$
- $\psi = \neg\psi$  and  $\alpha \not\models \psi$
- $\psi = \text{next}(\psi)$ ,  $\alpha[1] \models \psi$ , and  $n \geq 1$
- $\psi = \text{always}(\psi)$  and  $\forall (0 \leq i \leq n)$  we have that  $\alpha[i] \models \psi$
- $\psi = \text{eventually}(\psi)$  and  $\exists (0 \leq i \leq n)$  such that  $\alpha[i] \models \psi$
- $\psi = \text{until}(\psi_1, \psi_2)$  and  $\exists (0 \leq i \leq n)$  such that  $\alpha[j] \models \psi_1$  for all  $0 \leq j < i$  and  $\alpha[i] \models \psi_2$ .

where  $\varphi$  is a fluent formula and  $\psi_1$  and  $\psi_2$  are basic desires.

Let  $\psi$  be a basic desire and let  $\alpha$  and  $\beta$  be two trajectories. The trajectory  $\alpha$  is *preferred* to the trajectory  $\beta$  (denoted as  $\alpha \prec_\psi \beta$ ) if  $\alpha \models \psi$  and  $\beta \not\models \psi$ . When neither  $\alpha \prec_\psi \beta$  nor  $\beta \prec_\psi \alpha$  is the case, we say that  $\alpha$  and  $\beta$  are *indistinguishable* w.r.t.  $\psi$  and write  $\alpha \approx_\psi \beta$ .

**Atomic Preferences.** The definition of a basic desire implicitly implies that users have a set of desires in mind and their intention is to find a plan that satisfies all such desires. In many cases, this proves to be too strong and results in situations where no preferred plan can be found. Hence, it is necessary to provide users with a simple way to *rank* their basic desires, e.g., going by taxi is preferred to having coffee. Hence,  $\mathcal{PP}$  introduces another type of preferences, called *atomic preferences*, which allows the user to rank basic desires.

**Definition 3.** An atomic preference is a formula of the form  $\psi_1 \triangleleft \psi_2 \triangleleft \cdots \triangleleft \psi_n$  where  $\psi_1, \dots, \psi_n$  are basic desires.

It is easy to see that basic desires are special cases of atomic preferences — where all preference formulae have the same rank. The definitions of  $\approx$  and  $\prec$  between two trajectories  $\alpha$  and  $\beta$  w.r.t. a basic desire are extended to compare trajectories w.r.t. an atomic preference  $\psi = \psi_1 \triangleleft \psi_2 \triangleleft \cdots \triangleleft \psi_n$  as follows.  $\alpha$  and  $\beta$  are *indistinguishable* w.r.t.  $\psi$ , written as  $\alpha \approx_\psi \beta$ , if  $\alpha \approx_{\psi_i} \beta$  for all  $1 \leq i \leq n$ .  $\alpha$  is *preferred* to  $\beta$  w.r.t.  $\psi$ , written as  $\alpha \prec_\psi \beta$ , if there is  $1 \leq i \leq n$  s.t. (i)  $\alpha \approx_{\psi_j} \beta$  for  $1 \leq j < i$ , and (ii)  $\alpha \prec_{\psi_i} \beta$ .

**General Preferences.** Naturally, when a user has a set of atomic preferences, there is a need for combining them to create a new preference that can be used to select the best possible plan suitable to him/her. This is the concept of *general preference* in  $\mathcal{PP}$  and is defined as follows.

**Definition 4.** A general preference is either an atomic preference or a formula of the form  $\psi_1 \& \psi_2$ ,  $\psi_1 \mid \psi_2$ ,  $!\psi_1$ , or  $\psi_1 \triangleleft \psi_2$  where  $\psi_1$  and  $\psi_2$  are general preferences.

Note that the operators  $\&$ ,  $\mid$ ,  $!$  are syntactically similar to  $\wedge$ ,  $\vee$ ,  $\neg$  employed in the definition of basic desires. Semantically, they differ from these operations in a subtle way (see [13] for a discussion on this issue). The relations  $\prec$  and  $\approx$  between two trajectories  $\alpha$  and  $\beta$  w.r.t. a general preference  $\psi$  is defined as follows.

1.  $\alpha$  is *preferred* to  $\beta$ , denoted by  $\alpha \prec_{\psi} \beta$ , if one of the following holds.
  - (a)  $\psi$  is an atomic preference and  $\alpha \prec_{\psi} \beta$
  - (b)  $\psi = \psi_1 \& \psi_2$  and  $\alpha \prec_{\psi_1} \beta$  and  $\alpha \prec_{\psi_2} \beta$
  - (c)  $\psi = \psi_1 \mid \psi_2$  and either (i)  $\alpha \prec_{\psi_1} \beta$  and  $\alpha \approx_{\psi_2} \beta$ ; or (ii)  $\alpha \prec_{\psi_2} \beta$  and  $\alpha \approx_{\psi_1} \beta$ ; or (iii)  $\alpha \prec_{\psi_1} \beta$  and  $\alpha \prec_{\psi_2} \beta$ .
  - (d)  $\psi = !\psi_1$  and  $\beta \prec_{\psi_1} \alpha$
  - (e)  $\psi = \psi_1 \triangleleft \psi_2$  and  $\alpha \prec_{\psi_1} \beta$
  - (f)  $\psi = \psi_1 \triangleleft \psi_2$  and  $\alpha \approx_{\psi_1} \beta$  and  $\alpha \prec_{\psi_2} \beta$ .
2.  $\alpha$  is *indistinguishable* from  $\beta$ , denoted by  $\alpha \approx_{\psi} \beta$  if one of the following holds
  - (a)  $\psi$  is an atomic preference and  $\alpha \approx_{\psi} \beta$ .
  - (b)  $\psi = \psi_1 \& \psi_2$ ,  $\alpha \approx_{\psi_1} \beta$ , and  $\alpha \approx_{\psi_2} \beta$ .
  - (c)  $\psi = \psi_1 \mid \psi_2$ ,  $\alpha \approx_{\psi_1} \beta$ , and  $\alpha \approx_{\psi_2} \beta$ .
  - (d)  $\psi = !\psi_1$  and  $\alpha \approx_{\psi_1} \beta$ .
  - (e)  $\psi = \psi_1 \triangleleft \psi_2$ , and  $\alpha \approx_{\psi_1} \beta$  and  $\alpha \approx_{\psi_2} \beta$ .

### 3 Planning with Preferences as Constraint Satisfaction

Let  $\mathcal{P} = \langle D, I, G \rangle$  be a planning problem,  $\psi$  be a preference, and  $N$  be an integer. In this section, we first formulate  $\mathcal{P}$  as a constraint satisfaction problem  $\Pi(\mathcal{P}, N)$  whose satisfying assignments represent plans (of length  $N$ ) to  $\Pi$ . Then, we present an objective function for  $\Pi(\mathcal{P}, N)$  so that every satisfying assignment of  $\Pi(\mathcal{P}, N)$  with a maximal value of the objective function is a most preferred plan of  $\mathcal{P}^2$ .

Our transformation of  $\mathcal{P}$  into  $\Pi(\mathcal{P}, N)$  is similar to the SAT-based approach to planning such as [4, 12, 10]. For each fluent  $F$  in  $\mathbf{F}$ , we create a series of boolean variables, called *fluent variables*, in  $\Pi(\mathcal{P}, N)$  to denote the value of  $F$  at time  $0, 1, \dots, n$ . Similarly, each action  $a$  in  $\mathbf{A}$  is associated with a series of boolean variables, called *action variables*,  $a^i$  ( $0 \leq i \leq n-1$ ) to indicate whether  $a$  occurs at time  $i$ . The constraints of  $\Pi(\mathcal{P}, N)$  are divided into the following five groups.

1. **Initial state constraint.** The initial state is encoded as the following constraint in  $\Pi(\mathcal{P}, N)$ :

$$\left[ \bigwedge_{\text{initially}(l) \in I} l^0 \right] \wedge \left[ \bigwedge_{\text{caused}(p,l) \in D} (p^0 \rightarrow l^0) \right] \quad (5)$$

<sup>2</sup> Although the language  $\mathcal{PP}$  allows us to compare plans of different lengths, our goal in this paper is to find a most preferred plan among plans of the same length

The first (resp. second) conjunct says that the initial state must satisfy the initial state condition (resp. the static laws of the domain).

2. **Action occurrence constraints.** Since we are interested in finding sequential plans, the fact that exactly one action occurs at a time  $i$  is represented as the following constraint<sup>3</sup> ( $0 \leq i \leq n - 1$ )

$$\text{sum}_{a \in \mathbf{A}} a^i = 1 \quad (6)$$

3. **Executability constraints.** For each action  $a$ ,  $\Pi(\mathcal{P}, N)$  contains the following constraints ( $0 \leq i \leq n - 1$ ):

$$a^i \rightarrow \bigvee_{\text{executable}(a,p) \in D} p^i \quad (7)$$

That is to say, if action  $a$  occurs at time instant  $i$  then one of its executability conditions must hold at that time.

4. **Transition function constraints.** A transition  $\langle s_{i-1}, a_{i-1}, s_i \rangle$  is represented as the following constraints (for each fluent  $F$ ):

$$F^i \leftrightarrow \left[ \bigvee_{\text{causes}(a,F,p) \in D} (a^{i-1} \wedge p^{i-1}) \right] \vee \left[ \bigvee_{\text{caused}(p,F) \in D} p^i \right] \vee [F^i \wedge F^{i-1}] \quad (8)$$

$$\neg F^i \leftrightarrow \left[ \bigvee_{\text{causes}(a,\neg F,p) \in D} (a^{i-1} \wedge p^{i-1}) \right] \vee \left[ \bigvee_{\text{caused}(p,\neg F) \in D} p^i \right] \vee [\neg F^i \wedge \neg F^{i-1}] \quad (9)$$

The constraints say that a literal ( $F$  or  $\neg F$ ) holds in the successor state if either it is caused by a dynamic law (the first disjunct), or it is caused by a static law (the second disjunct), or it holds by the inertial law<sup>4</sup> (the last disjunct).

5. **Goal constraint.** The goal condition  $G$  is encoded as the constraint:

$$\bigwedge_{l \in G} l^N \quad (10)$$

It is proved in [10] that this encoding guarantees that every satisfying assignment of  $\Pi(\mathcal{P}, N)$  represents a plan to  $\mathcal{P}$ .

We now describe how to extend  $\Pi(\mathcal{P}, N)$  to handle preferences. Given a preference  $\psi$ , the basic idea to assign to each plan  $\pi$  a weight (an integer)  $w_\psi(\pi)$  in such a way that plans with a maximal weight are most preferred. Therefore, any satisfying assignment of  $\Pi(\mathcal{P}, N)$  maximizing the objective function  $w_\psi$  corresponds to a most preferred plan of length  $N$  of  $\mathcal{P}$ . We define such a function as follows.

- If  $\psi$  is a basic desire then (recall that  $\pi^*$  is the trajectory of  $\pi$ ):

$$w_\psi(\pi) = \begin{cases} 1 & \text{if } \pi^* \models \psi \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

- If  $\psi$  is an atomic preference  $\psi_1 \triangleleft \dots \triangleleft \psi_k$  then  $w_\psi(\pi) = \sum_{r=1}^k (2^{k-r} \times w_{\psi_r}(\pi))$
- Consider the case when  $\psi$  is a general preference.

<sup>3</sup> A boolean variable can also be viewed as an integer variable (*false* = 0 and *true* = 1)

<sup>4</sup> The inertial law says that normally a fluent remains unchanged.

1. if  $\psi$  is an atomic preference then  $w_\psi(\pi)$  is defined as in the previous case.
2. if  $\psi = \psi_1 \& \psi_2$  then  $w_\psi(\pi) = w_{\psi_1}(\pi) + w_{\psi_2}(\pi)$
3. if  $\psi = \psi_1 \mid \psi_2$  then  $w_\psi(\pi) = w_{\psi_1}(\pi) + w_{\psi_2}(\pi)$
4. if  $\psi = !\psi_1$  then  $w_\psi(\pi) = \max(\psi_1) - w_{\psi_1}(\pi)$  where  $\max(\psi_1)$  represents the maximum weight that a plan can achieve on  $\psi_1$  plus one.
5. if  $\psi = \psi_1 \triangleleft \psi_2$  then  $w_\psi(\pi) = \max(\psi_2) \times w_{\psi_1}(\pi) + w_{\psi_2}(\pi)$

It is proved in [13] that any plan with a maximal weight is a most preferred plan of length  $N$  of  $\mathcal{P}$ .

*Remark 1.* There may be many such weight functions. The weight function chosen is just one of them. We adopt it for the sake of easy computation.

*Remark 2.* Although each optimal solution of  $\Pi(\mathcal{P}, N)$  w.r.t. the function  $w_\psi$  is guaranteed to be a most preferred plan, the other direction does not hold in general. That is, some of the most preferred plans do not correspond to any optimal solution of  $\Pi(\mathcal{P}, N)$ . If we wish to find them, we might have to use another weight function.

## 4 Computing Preferred Plans using CLP

In this section, we describe a CLP based system, called CPP, to compute most preferred plans based on the formulation in the previous section. Although CPP was written in GNU Prolog [6], it can easily be ported to other Prolog systems that include a finite domain solver.

CPP takes as input a planning problem  $\mathcal{P} = (D, I, G)$ , an integer  $N$ , and a preference *Pref* and returns as output a most preferred plan  $\pi = a_0 a_1 \dots a_{N-1}$  of  $\mathcal{P}$  w.r.t. *Pref*. In our framework,  $\mathcal{P}$  is described in the language  $\mathcal{AL}$  as a Prolog program. As GNU Prolog does not allow the symbol  $\neg$ , a literal  $\neg F$  where  $F$  is a fluent is written as *neg*( $F$ ). Furthermore, in addition to propositions (1)-(4) for describing  $(D, I)$ , fluents and actions are explicitly declared using the sets of propositions  $\{\mathbf{fluent}(F) \mid F \in \mathbf{F}\}$  and  $\{\mathbf{action}(A) \mid A \in \mathbf{A}\}$ , and the goal  $G$  is described by the set of propositions  $\{\mathbf{goal}(L) \mid L \in G\}$ . Apart from this, because a basic desire or an atomic preference is just a special case of a general preference, without loss of generality, we assume that *Pref* is a general preference.

In CPP, each state  $s_i$  of  $\pi^*$  is represented as a list of fluent-variable pairs

$$s_i = [fluent(F_1, VarF_1^i), fluent(F_2, VarF_2^i), \dots, fluent(F_m, VarF_m^i)] \quad (12)$$

and each action  $a_i$  of  $\pi^*$  is represented as the following Prolog list

$$a_i = [action(A_1, VarA_1^i), action(A_2, VarA_2^i), \dots, action(A_l, VarA_l^i)] \quad (13)$$

where  $\{F_1, \dots, F_m\} = \mathbf{F}$  and  $\{A_1, \dots, A_l\} = \mathbf{A}$  ( $Var_j^i = 1$  denotes that action  $A_j$  occurs at time instant  $i^5$ ). The trajectory  $\pi^*$ , therefore, can be viewed as a pair of lists *Actions* and *States* where *Actions* is a list of representations of action occurrences (13) and *States* is a list of representations of states (12).

<sup>5</sup> We require that  $\sum_j VarA_j^i = 1$ , i.e., at any time exactly one action occurs

## 4.1 Main Procedure

The main procedure of the system is depicted in Figure 1. It takes as input a Prolog file *Domain* describing a planning problem  $\mathcal{P}$ , an integer  $N$ , and a preference *Pref* and returns as output a most preferred plan of  $\mathcal{P}$  w.r.t *Pref*. The set  $\mathbf{F}$  of fluent names and the set  $\mathbf{A}$  of action names are collected from the input planning problem  $\mathcal{P}$  and stored in lists *Lf* and *La* respectively, by `setof(F,fluent(F),Lf)` and `setof(A,action(A),La)`. Then, `make_states/3` and `make_actions/3`<sup>6</sup> use *La* and *Lf* to create the list of states *States* and the list of action occurrences *Actions*. In addition to creating action variables, `make_actions` also sets the requirement that exactly one action occur at time, i.e., constraint (6), using the GNU Finite Domain (FD) built-in constraint `fd_only_one/1`. Then, we collect the set of literals

```
main(Domain, Pref, N) :-
    consult(Domain), setof(F, fluent(F), Lf), setof(A, action(A), La),
    make_states(N, Lf, States), make_actions(N, La, Actions),
    setof(F, initially(F), Init), setof(F, goal(F), Goal),
    set_initial(Init, States), set_goal(Goal, States),
    transitions(Actions, States), set_execs(Actions, States),
    weight(Actions, States, Pref, W), collect_variables(States, Actions, Vars),
    fd_maximize(fd_labeling(Vars, [variable_method(standard),
    value_method(max)]), W), print_plan(Actions).
```

**Fig. 1.** Computing a most preferred trajectory

that hold in the initial state (propositions `initially(l)`) and the set of goal literals (propositions `goal(l)`) and store them in lists *Init* and *Goal* respectively. Predicates `set_initial/2` and `set_goal/2` (Section 4.2) set the constraints on the initial state and the goal state, respectively. Predicates `set_execs/2` and `transitions/2` (Section 4.3) set the executability and transition constraints for the trajectory. Next, the `main` procedure calls the `weight/4` predicate (Section 4.4) to compute the weight of  $\pi$  w.r.t. *Pref*. Specifically, `weight(Actions, States, Pref, W)` is equivalent to the constraint  $W = w_{Pref}(\pi)$ .

After setting up all necessary constraints on variables, the `main` procedure calls the GNU FD built-in predicate `fd_maximize/2` to find an assignment of variables that satisfies all the constraints and maximizes the weight of  $\pi$ , i.e., the *W*-value. Basically, `fd_maximize(Goal, X)` repeatedly calls `Goal` to find a value that maximizes the variable *X*. In our case, `Goal` is the use of `fd_labeling(Vars, [Options])`, where *Vars* is the set of all fluent and action variables collected by the function `collect_variables/3`, which assigns a value to each fluent variable or action variable so as to satisfy all the constraints according to labeling options *Options*. Our choices for labeling options are: the leftmost variable is chosen first `variable_method(standard)` and for each variable, the maximum value is enumerated first (`value_method(max)`). It is worth noting that the performance of

<sup>6</sup> Due to space limit, the code of some predicates is omitted.

the system depends on the order in which the variables are labeled. Our experiments showed that in most cases, labeling (fluent and action) variables backward yields the best performance. Hence, `collect_variables(Actions,States,Vars)` collects variables and sorts them in the descending order of time instants.

Finally, the `main` procedure calls the predicate `print_plan(Actions)` to print out the plan  $\pi$ . We now describe in more details the most important predicates `set_initial`, `set_goal`, `set_execs`, `transitions` and `weight` in CPP.

## 4.2 Initial State and Goal Constraints

The initial state and goal constraints (5) and (10) are asserted in CPP by predicates `set_initial/2` and `set_goal/2` respectively (see Figure 2). Both of these

```

set_initial(Init,[S|_]) :- set_state(Init,S),
    findall((P,L),caused(P,L),Stats), set_static_constraint(Stats,S).
set_static_constraint([],_).
set_static_constraint([(P,L)|Stats],S) :-
    holds(P,S,PrecHolds), holds(L,S,ConsHolds),
    PrecHolds ==> ConsHolds, set_static_constraint(Stats,S).
set_goal(Goal,States) :- last(States,S), set_state(Goal,S).
set_state([],_).
set_state([L|Lits],State) :- (L=neg(F),member(fluent(F,0),State);
    member(fluent(L,1),State)), set_state(Lits,State).

```

Fig. 2. Setting Initial State and Goal Constraints

predicates make calls to `set_state/2` to set the values for some fluents.

The first argument of `set_initial` is the list of literals that hold in the initial state and the second argument is the list of states *States*. First, `set_initial` calls `set_state` to set on the initial state *States*[0] the values for fluents (the first disjunct of (5)). Furthermore it calls the predicate `set_static_constraint/2` to establish static law constraints on the initial state (the second disjunct of (5)). The predicate `holds/3` is to test a literal, a list of literals, or a fluent formula holds in a state *S* (`holds(FF,S,R)` is equivalent to the constraint  $R \Leftrightarrow \text{“}FF \text{ holds in } S\text{”}$ , where *FF* can be either a literal, a list of literals or a fluent formula).

Likewise, `set_goal` calls the predicate `set_state` on the final state *States*[*N*] to establish the goal constraint (constraint (10)).

## 4.3 Executability and Transition Constraints

The system uses `set_execs/2` and `transitions/2` (Figure 3) to assert the constraints on the executability of actions (constraint (7)) and on the transitions of  $\pi$  (constraints (8) and (9)). At each iteration, `set_execs/2` calls the `set_exec/2` to assert the executability constraints at a specific time instant *i*. The first argument of `set_exec` is a set of pairs of action name – action variable for which executability constraints need be set and the second argument is the state at time instant

```

set_execs([], []).
set_execs([Action|Actions],[S|States]) :-
    set_exec(Action,S), set_execs(Actions,States).
set_exec([],_).
set_exec([action(A,VA)|Action],S) :- findall(P,executable(A,P),Precs),
    (Precs \== [] -> create_or(Precs,S,PrecHolds),
     VA #==> PrecHolds; true), set_exec(Action,S).
transitions(_Actions,[_States]) :- !.
transitions([Action|Actions],[S1,S2|States]) :-
    transition(Action,S1,S2,S1,S2), transitions(Actions,[S2|States]).
transition(_, [], [],_,_).
transition(Action,[fluent(F,V1)|LF1],[fluent(F,V2)|LF2],S1,S2):-
    set_one_fluent(F,V1,V2,Action,S1,S2), transition(Action,LF1,LF2,S1,S2).

set_one_fluent(F,V1,V2,Action,S1,S2) :-
    findall((A,DP),causes(A,F,DP),DPs), findall(SP,caused(SP,F),SPs),
    create_or(SP,S2,SPClause), dynamic_clause(Action,DPs,S1,DPCClause),
    (V2 #<=> (SPClause #\ DPCClause #\ (V1 #/\ V2)),
    findall(NegSP,caused(NegSP,neg(F)),NegSPs),
    findall((NegA,NegDP),causes(NegA,neg(F),NegDP),NegDPs),
    create_or(NegSPs,S2,NegSPClause),
    dynamic_clause(Action,NegDPs,S1,NegDPCClause),
    (#\ V2) #<=> (NegSPClause #\ NegDPCClause #\ (#\ V2 #/\ #\ V1)).
dynamic_clause(_, [],_,0).
dynamic_clause(Action,[(A,P)|T],S1,R) :-
    happen(A,Action,VA), holds(P,S1,PrecHolds),
    dynamic_clause(Action,T,S1,R1), R #<=> (R1 #\ (VA #/\ PrecHolds)).

```

Fig. 3. Setting Executability Constraints and Transition Constraints

*i.* `set_exec(action(A,VA),S)` first finds all the executability conditions (4) for action  $A$ , stores them in a list  $Precs$  and then calls the predicate `create_or/3`<sup>7</sup> to create the disjunction  $PrecHolds$  for the right hand side of constraint (7) if at least one such executability condition exists. Finally, constraint (7) is enforced by  $VA\# ==> PrecHolds$ . When action  $A$  has no executability condition then nothing is set because by default we assume that  $A$  is executable.

The transition constraints (8)-(9) are asserted in CPP by `transitions/2` (Figure 3). This predicate repeatedly calls `transition/5` to set the transition constraints at each time instant. Basically, `transition(Action,List1,List2,S1,S2)` set constraints (8)-(9) for a set of fluents whose values in states  $S1$  and  $S2$  are stored in  $List1$  and  $List2$  respectively. The main part of `transition` is the call to `set_one_fluent/6`. Intuitively, `set_one_fluent(F,V1,V2,Action,S1,S2)`, where  $F$  is a fluent,  $\langle S1, Action, S2 \rangle$  is a transition, and  $V1$  and  $V2$  are the values of  $F$  in  $S1$  and  $S2$  respectively, sets constraints (8)-(9) on literals  $F$  and  $\neg F$ . This predicate calls `create_or` and `dynamic_clause` to generate the first and second disjuncts of the right side of (8)-(9).

<sup>7</sup> `create_or(Ps,S,R)` is equivalent to the constraint  $R \Leftrightarrow \bigvee_{p \in Ps} p$  holds in  $S$

#### 4.4 Handling Preferences

We now describe how to compute the weight of  $\pi$  with respect to *Pref*, i.e., the weight predicate. Recall that in CPP  $\pi^*$  is represented by the list of action occurrences *Actions* and the list of states *States*.

```

satisfy(Actions,States,T,goal(FF),W) :-
  length(States,N), T=<N, !, satisfy(Actions,States,N,FF,W).
satisfy(Actions,States,T,occ(A),W) :-
  length(States,N), T<N, !, nth(T,Actions,Action), happen(A,Action,W).
satisfy(Actions,States,T,and(D1,D2),W) :-
  length(States,N), T=<N, !, satisfy(Actions,States,T,D1,W1),
  satisfy(Actions,States,T,D2,W2), W#<=>(W1+W2#=#2).
satisfy(Actions,States,T,or(D1,D2),W) :- length(States,N), T=<N, !,
  satisfy(Actions,States,T,D1,W1), satisfy(Actions,States,T,D2,W2),
  W#<=>(W1+W2#>0).
satisfy(Actions,States,T,neg(D),W) :-
  length(States,N), T=<N, !, satisfy(Actions,States,T,D,W1), W#=#1-W1.
satisfy(Actions,States,T,eventually(D),W) :-
  length(States,N), T=<N, !, satisfy(Actions,States,T,D,W1), T1 is T+1,
  satisfy(Actions,States,T1,eventually(D),W2), W#<=>(W1+W2#>0).
satisfy(Actions,States,T,next(D),W) :-
  length(States,N), T=<N, !, T1 is T+1, satisfy(Actions,States,T1,D,W).
satisfy(Actions,States,T,always(D),W) :-
  length(States,N), T=<N, !, during(Actions,States,T,N,D,W).
satisfy(Actions,States,T,until(D1,D2),W) :-
  length(States,N), T=<N, !, satisfy(Actions,States,T,D2,W1),
  satisfy(Actions,States,T,D1,W2), T1 is T+1,
  satisfy(Actions,States,T1,until(D1,D2),W3), (W#<=>(W1#\/(W2+W3#=#2))).
satisfy(_,States,T,D,W) :-
  is_formula(D), length(States,N), T=<N, !, nth(T,States,S), holds(D,S,W).
satisfy(_,States,T,D,0) :- length(States,N), (D=occ(_), N=<T; N>T).

during(Actions,States,T1,T2,D,W) :-
  T1 < T2, !, satisfy(Actions,States,T1,D,W1), T3 is T1+1,
  during(Actions,States,T3,T2,D,W2), W#<=>(W1+W2#=#2).
during(Actions,States,T,T,D,W) :- !, satisfy(Actions,States,T,D,W).

```

**Fig. 4.** Checking satisfaction of a basic desire.

As have been seen in Section 3, in order to compute the weight of  $\pi$  w.r.t. a basic desire  $D$ , it is crucial to check if  $\pi^*$  satisfies  $D$  (see Eq. (11)). In CPP, the predicate `satisfy/5` (Figure 4) is devoted to doing this task. Intuitively, `satisfy(Actions,States,T,D,W)` sets  $W$  to 1 if desire  $D$  is satisfied by the trajectory of  $\pi^*$  starting at time instant  $T$ ,  $\pi^*[T]$ , and to 0 if otherwise. `satisfy` is a quite straightforward implementation of the satisfaction  $\models$  in Definition 2. Note that the implementation of `satisfy` makes calls to predicates `holds/3`, `is_formula/1`, `happen/3`, and `during/6`. The first predicate, already mentioned

before, is to check if a fluent formula holds in a certain state. The second is to check if an expression is a valid fluent formula. `happen(A,Action,W)` is equivalent to  $W \Leftrightarrow$  “ $A$  occurs at time instant  $i$ ”, where  $Action$  is the representation of action occurrence at time step  $i$ . Finally, `during(Actions,States,T1,T2,D,W)` sets  $W$  to 1 if every trajectory  $\pi^*[i]$  where  $T1 \leq i \leq T2$  satisfies  $D$  and, to 0 if otherwise.

Figure 5 shows the implementation of the `weight` predicate based on the predicate `satisfy`. Intuitively, `weight(Actions,States,Pref,W)` sets  $W$  to  $w_{Pref}(\pi)$ . The `max/2` predicate is the implementation of the *max*-function which computes the maximum weight that a plan can achieve on  $Pref$  plus one (see Section 3). Note that `power(N,W)` is equivalent to  $W = 2^N$ .

```
weight(_,_,[],0).
weight(Actions,States,[D|Ds],W) :-
    satisfy(Actions,States,1,D,W1), length(Ds,NE),
    power(NE,COEF), weight(Actions,States,Ds,W2), W#=COEF*W1+W2.
weight(Actions,States,and(Pref1,Pref2),W) :-
    weight(Actions,States,Pref1,W1),weight(Actions,States,Pref2,W2),W#=W1+W2.
weight(Actions,States,or(Pref1,Pref2),W) :-
    weight(Actions,States,Pref1,W1),weight(Actions,States,Pref2,W2),W#=W1+W2.
weight(Actions,States,neg(Pref1),W) :-
    weight(Actions,States,Pref1,W1), max(Pref1,W2), W#=W2-W1.
weight(Actions,States,prec(Pref1,Pref2),W) :-
    weight(Actions,States,Pref1,W1), weight(Actions,States,Pref2,W2),
    max(Pref2,W3), W #= W3*W1+W2.

max([H|T],W) :- length([H|T],N), power(N,W).
max(and(Pref1,Pref2),W) :- max(Pref1,W1), max(Pref2,W2), W is W1+W2.
max(or(Pref1,Pref2),W) :- max(Pref1,W1), max(Pref2,W2), W is W1+W2.
max(neg(Pref1),W) :- max(Pref1,W).
max(prec(Pref1,Pref2),W) :- max(Pref1,W1), max(Pref2,W2), W is W1*W2+W1.
```

**Fig. 5.** Implementation of the weight function

Let us describe in more details how the `weight` predicate is implemented. Since we are assuming that  $Pref$  is a general preference, there are five cases.

1.  $Pref$  is an atomic preference. In CPP, an atomic preference is represented as a Prolog list of basic desires  $[D1, D2, \dots]$ . Then the implementation of `weight` for this particular case is based on `satisfy` above. First if  $Pref$  is an empty list then the weight of  $\pi$  w.r.t.  $Pref$  is 0. If it is of the form  $[D|Ds]$  then the weight of  $\pi$  w.r.t.  $Pref$  is  $2^{NE} * W1 + W2$  where  $NE$  is the length of  $Ds$ ,  $W1$  is the weight of  $\pi$  w.r.t. the basic desire  $D$  and  $W2$  is the weight of  $\pi$  w.r.t.  $Ds$ .
2.  $Pref = Pref1 \& Pref2$  where  $Pref1$  and  $Pref2$  are general preferences. Then, the weight of  $\pi$  w.r.t.  $Pref$  is  $W = W1 + W2$  where  $W1$  and  $W2$  are the weight of  $\pi$  w.r.t.  $Pref1$  and the weight of  $\pi$  w.r.t.  $Pref2$ .
3.  $Pref = Pref1 | Pref2$  where  $Pref1$  and  $Pref2$  are general preferences. Then, the weight of  $\pi$  w.r.t.  $Pref$  is defined as the same as in the previous case.

4.  $Pref = !Pref1$  where  $Pref1$  is a general preference. In this case, the weight of  $\pi$  w.r.t.  $Pref$  is  $W = W2 - W1$ , where  $W2$  is the maximal weight of a plan that can be achieved on  $Pref1$  and  $W1$  is the weight of  $\pi$  w.r.t.  $Pref1$ .
5.  $Pref = Pref1 \triangleleft Pref2$ . In this case, the weight of  $\pi$  is  $W = W1 * W3 + W2$ , where  $W1$ ,  $W2$ , and  $W3$  is the weight of  $\pi$  w.r.t.  $Pref1$ , the weight of  $\pi$  w.r.t.  $Pref2$  and the maximal weight of a plan that can be achieved on  $Pref2$ .

## 5 Experiments

To evaluate<sup>8</sup> the performance of CPP, we tested it on two domains: the infamous blocks world domain (Block) and the rocket domain (Rocket)[15].

In the Block domain, there are  $m \times n$  blocks, numbered from  $1..m \times n$ . Initially, blocks are organized in  $m$  piles, each having  $n$  blocks. The  $i$ -th pile consists of  $n$  blocks that are numbered with  $(1 + (i - 1) \times n) \dots (i \times n)$  where the blocks with smaller numbers are on top of the blocks with larger numbers. The goal is to have a pile that consists of blocks from  $1$  to  $m \times n - 1$  where the blocks with larger numbers are on top of the blocks with smaller numbers. The position of the remaining block, i.e., block number  $m \times n$ , can be anywhere (either on the table, or on block  $m \times n - 1$  or under block 1). In our experiments we tested instances with  $m = 1$  and  $n$  from 4..7 and with  $m = 2$  and  $n = 3, 4$ . We ran these instances with preferences  $\psi_1, \dots, \psi_8$  defined as follows. The first four preferences are basic desires.  $\psi_1$  is a goal preference of having the last block (block number  $m \times n$ ) on the top of block  $m \times n - 1$  in the final state.  $\psi_2$  is also a goal preference but it prefers to have the last block under block 1.  $\psi_3$  and  $\psi_4$  are state desires.  $\psi_3$  says that never place any block except block 1 on the table;  $\psi_4$  says that at sometime block 1 is on block  $m \times n$ .  $\psi_5$  is the atomic preference  $\psi_1 \triangleleft \psi_2$  and  $\psi_6$  is the atomic preference  $\psi_3 \triangleleft \psi_4$ .  $\psi_7$  and  $\psi_8$  are general preferences  $\psi_4 \& \psi_5$  and  $\psi_8 = \psi_4 | \psi_5$  respectively.

In the Rocket domain, we need to move a cargo by a rocket from one location to another location. There are  $n$  locations  $l_1, l_2, \dots, l_n$  and initially the rocket is at the first location and the goal is to have it at the last location. We did experiments with  $n = 3, 4, 5$  and preferences  $\psi_1, \dots, \psi_7$  as follows. The first three are basic preferences:  $\psi_1$  is a goal preference of having the rocket at the first location in the final state;  $\psi_2$  prefers that the rocket visits all other locations;  $\psi_3$  prefers the rocket never reaches a location  $l_i$  before a location  $l_j$  where  $i > j$ .  $\psi_4$  and  $\psi_5$  are atomic preferences  $\psi_1 \triangleleft \psi_2$  and  $\psi_2 \triangleleft \psi_3$  respectively.  $\psi_6$  and  $\psi_7$  are general preferences  $! \psi_4$  and  $\psi_4 \triangleleft \psi_5$ .

All experiments were run on a 2.4 GHz CPU, 768MB RAM machine. The version of the GNU Prolog system used for testing is GNU Prolog 1.2.16 and was downloaded from <http://gnu-prolog.inria.fr/>. Time out is set to 10 minutes. The test results are shown in Tables 1 – 3. In these tables,  $N$  is the length of plans that we wish to find; each cell shows the time for CPP to return a most preferred plan and the weight of the plan; times are shown in seconds; TO indicates a timeout.

As can be seen from Tables 1-2, the performance of CPP on the block world domain is relatively good. When the number of blocks is less than or equal to

<sup>8</sup> The source code of CPP and testing domains can be found at <http://www.cs.nmsu.edu/~tphan/software.htm>

Domain	N	$\psi_1$	$\psi_2$	$\psi_3$	$\psi_4$	$\psi_5$	$\psi_6$	$\psi_7$	$\psi_8$
Block(1,4)	4	0.00/1	0.00/0	0.00/1	0.00/0	0.00/2	0.00/2	0.00/4	0.00/4
	5	0.02/1	0.00/0	0.00/1	0.00/0	0.02/2	0.02/2	0.02/4	0.02/4
	6	0.00/1	0.00/1	0.02/1	0.02/1	0.00/2	0.03/2	0.02/4	0.02/4
	7	0.02/1	0.00/1	0.02/1	0.03/1	0.00/2	0.03/2	0.03/4	0.03/4
Block(1,5)	5	0.00/1	0.02/0	0.02/1	0.02/0	0.00/2	0.02/2	0.02/4	0.02/4
	6	0.02/1	0.03/0	0.03/1	0.02/0	0.02/2	0.05/2	0.05/4	0.03/4
	7	0.03/1	0.03/0	0.05/1	0.06/0	0.03/2	0.09/2	0.05/4	0.06/4
	8	0.08/1	0.03/1	0.08/1	0.08/1	0.06/2	0.14/2	0.06/4	0.08/4
Block(1,6)	6	0.02/1	0.02/0	0.31/1	0.02/0	0.02/2	0.03/2	0.03/4	0.03/4
	7	0.06/1	0.08/0	0.16/1	0.09/0	0.09/2	0.17/2	0.14/4	0.14/4
	8	0.11/1	0.14/0	0.19/1	0.28/0	0.14/2	0.31/2	0.22/4	0.24/4
	9	0.42/1	0.39/0	0.28/1	0.56/0	0.47/2	0.52/2	0.66/4	0.67/4
	10	2.00/1	0.28/1	0.53/1	0.66/0	2.33/2	0.94/2	2.02/4	2.08/4
Block(1,7)	9	0.48/1	0.47/0	0.67/1	0.78/0	0.5/2	0.97/2	0.52/4	0.52/4
	10	1.94/1	1.5/0	0.86/1	1.81/0	2.28/2	1.55/2	1.72/4	1.72/4
	11	10.78/1	6.00/0	1.97/1	7.27/0	11.02/2	2.81/2	7.64/4	7.5/4
	12	66.28/1	3.34/1	2.61/1	5.3/1	76.24/2	5.08/2	40.67/4	41.31/4

**Table 1.** Performance of CPP on the blocks world domain

Domain	N	$\psi_1$	$\psi_2$	$\psi_3$	$\psi_4$	$\psi_5$	$\psi_6$	$\psi_7$	$\psi_8$
Block(2,3)	5	0.2/1	0.23/0	0.36/1	0.38/0	0.24/0	0.55/2	0.56/2	0.66/2
	6	0.12/1	0.18/0	0.2/1	0.47/0	0.11/2	0.44/2	0.22/4	0.2/4
	7	0.25/1	0.31/1	0.23/1	0.75/1	0.27/2	0.38/2	0.34/4	0.33/4
	8	0.82/1	1.15/1	0.28/1	1.92/1	0.81/2	0.89/2	1.05/4	1.05/4
Block(2,4)	7	3.95/0	4.28/0	3.06/1	5.69/0	7.84/2	5.12/2	5.86/2	5.82/2
	8	1.26/1	1.72/0	1.08/1	5.51/0	2.15/2	2.65/2	1.4/4	1.4/4
	9	7.2/1	7.75/0	1.34/1	27.56/0	8.75/2	4.97/2	4.93/4	4.98/4
	10	44.58/1	33.75/1	1.58/1	43.42/1	38.79/2	8.95/2	27.15/4	27.12/4

**Table 2.** Performance of CPP on the blocks world domain

6 (problems *Block(1,4)* & *Block(1,5)*), the solving time is negligible (less than 0.1s). However, when the number of blocks increases to more than 6 (instances *Block(1,6)*, *Block(1,7)*, *Block(2,3)*, and *Block(2,4)*), the solving time increases exponentially. For the Rocket domain (Table 3), within the time limit, CPP could solve all instances of Rocket(3) and Rocket(4). However, for the Rocket(5) instance, in most cases, CPP reported a time out. On the other hand, it is noticeable that for each instance (either a blocks world instance or a rocket instance) when the input parameter  $N$  increases, the solving time is exponentially increases. This is because of the blow-up in the number of variables in the program.

Domain	N	$\psi_1$	$\psi_2$	$\psi_3$	$\psi_4$	$\psi_5$	$\psi_6$	$\psi_7$
Rocket(3)	3	0.00/0	0.00/0	0.00/0	0.00/0	0.02/0	0.01/4	0.01/0
	4	0.02/1	0.03/1	0.03/1	0.01/2	0.03/3	0.02/3	0.02/8
	5	0.05/1	0.09/1	0.11/1	0.08/3	0.12/3	0.08/4	0.13/15
Rocket(4)	4	0.02/1	0.19/0	0.31/0	0.23/2	0.22/0	0.22/4	0.27/8
	5	1.08/1	2.17/1	2.39/1	1.66/2	3.56/3	1.64/4	1.95/8
	6	7.64/1	14.81/1	15.81/1	15.22/3	24.37/3	11.83/4	21.34/15
Rocket(5)	5	17.39/1	22.05/0	38.95/1	28.65/2	44.87/1	28.84/4	33.75/8
	6	239.94/1	400.34/1	TO	TO	50.38/3	TO	TO
	7	TO	TO	TO	TO	TO	TO	TO

**Table 3.** Performance of CPP on the rocket domain

## 6 Conclusion and Future Work

In this paper, we formulate the planning problem with preferences as a constraint satisfaction problem. We then present the development of a CLP based system, called CPP, which is capable of generating most preferred plans with respect to a user's preference and show experimental results. At present CPP is implemented in GNU prolog. It is worth noting that there are many other different constraint

programming systems and the performance of a constraint program heavily depends on the encoding of the problem and on the underlying solver. Hence, as future work, on the one hand, we would like to try different encodings of CPP on different systems. On the other hand, we would like to investigate the usefulness of heuristics and the applicability of constraint handling rules [8] to improve the performance of CPP. In addition, we would like to extend CPP to be able to deal with non-deterministic and/or incomplete action theories. This involves extending the preference language  $\mathcal{PP}$  so as to be able to compare plans in non-deterministic and/or incomplete action theories.

## References

1. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In Minker, J., ed.: *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers (2000) 257–279
2. Bienvenu, M., Fritz, C., McIlraith, S.: Planning with qualitative temporal preferences. In *KR*, Lake District, UK (2006)
3. Brafman, R.I., Chernyavsky, Y.: Planning with goal preferences and constraints. In *ICAPS*, (2005) 182–191
4. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based Planning in Complex Domains: Concurrency, Constraints and Nondeterminism. *AI* **147**(1-2) (2003) 85–117
5. Delgrande, J.P., Schaub, T., Tompits, H.: Domain-specific preferences for causal reasoning and planning. In *KR*, (2004) 673–682
6. Diaz, D., Codognot, P.: Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming* **2001**(6) (2001)
7. Dovier, A., Formisano, A., Pontelli, E.: Planning with Action Languages: Perspectives using CLP(FD) and ASP. In *CILC*. (2006)
8. Frühwirth, T.: Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming* **37**(1-3) (1998) 95–138
9. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia (2005)
10. Giunchiglia, E., Lifschitz, V.: An action language based on causal explanation: preliminary report. In: *Proceedings of AAAI 98*. (98) 623–630
11. Jaffar, J., Maher, M.: Constraint Logic Programming. *JLP* **19/20** (1994)
12. Kautz, H., Selman, B.: Planning as satisfiability. In: *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*. (1992) 359–363
13. Son, T.C., Pontelli, E.: Planning with Preferences using Logic Programming. *Theory and Practice of Logic Programming* (2006) To Appear.
14. Van Hentenryck, P.: *Constraint Satisfaction in Logic Programming*. MIT Press (1989)
15. Veloso, M.: Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University (1989)