UCPOP User's Manual

(Version 2.0)

Anthony Barrett, Keith Golden, Scott Penberthy & Daniel Weld

Technical Report 93-09-06
January 27, 1994

Department of Computer Science and Engineering[1]
University of Washington
Seattle, WA 98105
bug-ucpop@cs.washington.edu

[1]We thank Marc Young for contributions to the documentation, Claudia Chiang and Alan Lundy for comments and proofreading, and other members of the AI group for discussions and suggestions regarding UW planning research. This work was funded in part by National Science Foundation Grant IRI-8957302, Office of Naval Research Grant 90-J-1904, and a grant from the Xerox corporation.

Abstract

The UCPOP partial order planning algorithm handles a subset of the KRSL action representation [?] that corresponds roughly with Pednault's ADL language [?] and PRODIGY's PDL language [?]. In particular, UCPOP operates with actions defined using conditionals and nested quantification. This manual describes a COMMON LISP implementation of UCPOP, explains how to set up and run the planner, details the action representations syntax, and outlines the main data structures and how they are manipulated. Improvements in version 2.0 include:

- Declarative specification of search control rules
- Universal quantification over dynamic universes (*i.e.*, object creation and destruction)
- Domain axioms
- Predicates expanding to lisp code
- Larger set of domain theories search functions for testing.
- Expanded users manual.
- Improved CLIM-based graphic plan-space browser

Contents

1 Introduction

This handout details the UCPOP planning system — a clean COMMON LISP implementation of an elegant algorithm for partial order planning with an expressive action representation. UCPOP handles a large subset of ADL [?], including actions with conditional effects, universally quantified preconditions and effects, and universally quantified goals. UCPOP has desirable formal properties: *e.g.*, [?] proves soundness and completeness. Since UCPOP's simplicity and efficient implementation make it an excellent vehicle for further research on planning and learning, we provide this description of the implementation.

1.1 The UCPOP Algorithm

UCPOP represents a planning problem with an initial, dummy plan that consists solely of a "start" step (whose effects encode the initial conditions) and a "goal" step (whose preconditions encode the goals). UCPOP then attempts to complete this initial plan by adding new steps and constraints until all preconditions are guaranteed to be satisfied. The main loop makes two types of choices: supporting "open" preconditions and resolving "threats."

- If UCPOP has not yet satisfied a precondition (*i.e.*, it is "open"), then all step effects that could possibly be constrained to unify with the desired proposition are considered. UCPOP chooses one effect nondeterministically[2] and then adds a "causal link" to the plan to record this choice.
- If a third step S_k (called a "threat") might possibly interfere with the precondition being supported by a causal link from S_i to S_j, UCPOP nondeterministically protects against this threat by performing one of

  1. Promotion: if consistent, move S_k after S_j; or
  2. Demotion: if consistent, move S_k before S_i; or
  3. Separation: if consistent, add binding constraints to ensure that S_k cannot interfere with r; or
  4. Confrontation: if the threat results from a conditional effect, add a new subgoal that negates the preconditions of the offending effect.

Once UCPOP has successfully created and protected a causal link for every goal in the plan, it halts and returns a solution. Any totally ordered completion of this partially ordered plan will achieve the problem's goals.

Of necessity, this overview has been brief and conceptual. For a complete description of the UCPOP algorithm see the file `kr92-ucpop.ps` which contains a postscript version of the definitive paper: [?].

[2] In fact, domain dependent information can be used to guide the choice. Backtracking ensures that all choices will be eventually considered.

1.2 New Features

Since version 1.00a many features have been added to UCPOP. First, the domain definition language is enhanced to improve quantification and handle dynamic universes — action effects can create new objects or delete existing ones and quantified goals will work properly.

UCPOP 2.0 now supports domain axioms as explained in section 4.4. For tractability reasons, the domain-axiom facility partitions predicates into derived and primitive classes. Actions can only affect primitive terms while axioms are restricted to assert derived predicates. Conceptually, after an action is executed, the world state is updated by asserting an action's effects and then firing all domain axioms until none apply. Of course, since UCPOP does backward chaining, it doesn't actually implement this conceptual model, but the model suggests how one should use domain axioms: to define terms (like `clear` and `above`) in terms of primitive predicates (such as `on`).

Section 4.5 shows how to define predicates that call user defined lisp functions when used as preconditions. This facility greatly extends the practical utility of UCPOP when dealing with arithmetic etc.

Perhaps the most important addition to UCPOP is the search control facility. Many users complained that the ranking functions were a cumbersome way to direct the search through planning space. Version 2.0 allows users to define a rule based search controller to guide UCPOP's nodeterministic choices. Declarative rules allow much more flexible control of search; see section 5 for the details.

Note that the operator definition syntax is different (improved to be more regular), but all old domain definitions will still work; see section 4 for the details. The graphic plan-space browser has also been extended, but basic operation is essentially the same.

1.3 Document Overview

Section 2 explains installation and loading. Section 3 shows how to call the planner on existing examples, lists the basic interface, and discusses the X window-based search space browser which uses a VCR metaphor for navigation. Section 4 describes the UCPOP action language in detail and explains how to define new domains. Section 6 lists some of the primitive calls and switches that UCPOP supports. Finally, section 7 provides an overview of UCPOP's data structures for users that wish to extend the planner.

2 Installing and Loading UCPOP

The core UCPOP system consists of the following files:

`struct.lisp`	Basic data structure definitions
`variable.lisp`	Unification & codesignation constraint handling
`ucpop.lisp`	Actual planning algorithm
`plan-utils.lisp`	Routines for testing domains and creating steps
`vcr.lisp`	Optional X-window graphical interface
`choose.lisp`	Routines for computing preference orders
`rules.lisp`	General production system
`scr.lisp`	Rule based search controller

`interface.lisp`	Top level interface for calling UCPOP
`domains.lisp`	Example domain and problem definitions
`controllers.lisp`	Example search controller definitions for UCPOP

In addition, the file `ucpop.system` defines the dependencies necessary to enable use of Mark Kantrowitz's public domain "Portable Mini-DefSystem." Once you have installed the system file[3] you need only type:

```
(require 'ucpop)
```

to load the planner. Alternatively, you may load UCPOP by editing the file `loader.lisp` so as to change the value of `*ucpop-dir*` to the directory where you have copied these files. Then start lisp and load `loader.lisp` and type `(compile-ucpop)`. In future interactions, you need only type `(load-ucpop)` to load the compiled version.

Note that the graphical interface (implemented in `vcr.lisp` and described in section 3.4) requires CLIM, the COMMON LISP INTERFACE MANAGER, for operation. Regardless of how it is loaded, UCPOP checks to see if CLIM exists (by looking for the `clim` package) and only includes `vcr` if appropriate.

3 Operating UCPOP

Once you have loaded UCPOP, type:

```
(in-package "UCPOP")
```

to enter the UCPOP package. The rest of this manual assumes that you are in this package and refers to symbols locally.

3.1 A Simple Example

To run a small example from `domains.lisp` you can type

```
(bf-control 'uget-paid)
```

This defines and runs Pednault's famous example [**?**] involving transportation of objects between home and work using a briefcase whose effects involve both universal quantification (*all* objects are moved) and conditional effects (*if* they are inside the briefcase when it is moved). Section 1.1 of [**?**] describes how UCPOP solves a problem in this domain. You may find it helpful to refer to that paper as you read the actual COMMON LISP encoding below. The domain is described in terms of three action schemata:

```
(define (operator mov-b)
  :parameters (?m ?l)
  :precondition (and (at B ?m) (neq ?m ?l))
  :effect (and (at b ?l) (not (at B ?m))
               (forall (?z)
                   (when (and (in ?z) (neq ?z B))
                      (and (at ?z ?l)  (not (at ?z ?m)))))) )
```

[3] See the `defsystem` documentation for instructions on installing a central system registry and using `defsystem`. Contact `mkant@cs.cmu.edu` for information on acquiring the public domain code.

This specifies that the briefcase can be moved from location `?m` to `?l` where the symbols starting with question marks denote variables. The preconditions dictate that the briefcase must initially be in the starting location for the action to be legal and that it is illegal to try to move the briefcase to the place where it is initially. The effect equation says that the briefcase moves to its destination, is no longer where it started, and everything inside the briefcase is likewise moved.

```
(define (operator put-in)
    :parameters (?x ?l)
    :precondition (neq ?x B)
    :effect (when (and (at ?x ?l) (at B ?l))
                (in ?x)) )
```

This operator specifies the effect of putting something (not the briefcase (`B`) itself!) inside the briefcase. If the action is attempted when the object is not at the same place (`?l`) as the briefcase, then there is no effect.

```
(define (operator take-out)
    :parameters (?x)
    :precondition (neq ?x B)
    :effect (not (in ?x)) )
```

The final operator provides a way to remove something from the briefcase. Pednault's example problem supposed that at home one had a dictionary and a briefcase with a paycheck inside it. This situation can be specified with the list of true facts:

```
'((at B home) (at P home) (at D home) (in P))
```

because UCPOP assumes that all facts not declared to be true are false. Suppose that we wished to have the dictionary and briefcase at work, but wanted to keep the paycheck at home. We could write this desire as the conjunction:

```
'(and (at P home) (at D office) (at B office))
```

Note that initial conditions and effects are not completely symmetric! Although the initial conditions are all true together (*i.e.*, they are a conjunct), this conjunctive nature is implicit — you don't need to say `and`. Instead the initial conditions are specified with a simple list of terms. Goals on the other hand (even if they are a simple conjunction) must have the `and` included explicitly! This is because one can specify other types of goals besides simple conjunctions (see the definitions is section 4.2).

The file `domains.lisp` defines this planning problem and gives it the name `uget-paid`. Thus when one gives the `bf-control` command listed at the beginning of this section, UCPOP will find and display a plan to solve the problem. In this case, it prints:

```
Initial  : ((AT B HOME) (AT P HOME) (AT D HOME) (IN P))

Step 1   : (PUT-IN D HOME)        Created 3
```

```
           0  -> (AT D HOME)
           0  -> (AT B HOME)
Step 2  : (TAKE-OUT P)           Created 2
Step 3  : (MOV-B HOME OFFICE)    Created 1
           3  -> (IN D)
           0  -> (AT B HOME)
           2  -> (NOT (IN P))

Goal    : (AND (AT B OFFICE) (AT D OFFICE) (AT P HOME))
           1  -> (AT B OFFICE)
           1  -> (AT D OFFICE)
           0  -> (AT P HOME)
Complete!

UCPOP Stats: Initial terms = 4 ;   Goals = 4 ;  Success (3 steps)
      Created 42 plans, but explored only 22
      CPU time:    0.1340 sec
      Branching factor:  1.455
      Working Unifies: 248
      Bindings Added: 63
#plan<S=4; O=0; U=0>
#Stats:<cpu time = 0.1340>
```

Here we see the correct steps in order. Although UCPOP only inferred late in the planning process (*i.e.*, after creating the first two steps) that it needed to put the dictionary in the briefcase, it correctly deduced that this step should come early in the plan. Also, since it doesn't matter whether the dictionary is put in the briefcase before the paycheck is removed, UCPOP will not order these two steps, but the plan display routine chooses a (legal) total order for clarity.

Underneath most steps are a sequence of lines with a number followed by a `->` followed by a term. These lines list the producing step identifiers for each of the preconditions of the current step. For example, under the goal "step" are three lines and the first indicates that the step which was created first (*i.e.*, the `move-b` action) is responsible for achieving the first goal proposition(*i.e.*, `(at b office)`).

The last few lines printed by the `bf-control` command give myriad statistics showing how well UCPOP performed on this problem. The function returns a plan and a statistics structure. These structures are tersely printed at the bottom.

3.2 Predefined Problems

As mentioned in the last section, the file `domains.lisp` defines a number of named domains and problems. It records these problems using the global variable `*tests*` which is assumed to be a list of `problem` structures. Each problem has five fields:

- `name` This is just a symbol, like `uget-paid` to be passed to `bf-control` as an argument identifying the problem.

- `domain` This is a function which when called will initialize the domain operators.
- `inits` This is a list of terms, usually propositions that are true in the world.
- `goal` This can be a complicated logical expression with conjunction, disjunction, and quantification. In the example above we saw a simple case, but the formal definition of what is allowed is defined as a `GD` (goal description) in the EBNF description of section 4
- `rank-fun` An optional plan ranking function to guide search. If not specified, it defaults to the standard domain-independent ranking function. See section 6 for details on writing new ranking functions.

For example, assuming that the function `blocks-world-domains` had already been defined, we could specify the Sussman anomaly with the following expression:

```
(define (problem sussman-anomaly)
    :domain #'blocks-world-domain
    :inits ((block A) (block B) (block C) (block Table)
            (on C A) (on A Table) (on B Table)
            (clear C) (clear B) (clear Table))
    :goal (and (on B C) (on A B)))
```

Since this piece of code has already been defined (as part of `domains.lisp`) you can see how UCPOP handles the Sussman anomaly by calling

```
(bf-control 'sussman-anomaly)
```

The output should resemble the comment section in `domains.lisp`. Take some time to look through this file for other interesting domains and problems. UCPOP works on most, but not on all of these examples.

3.3 Bounding Search

In some cases, UCPOP will exceed its resource bounds before finding a solution. When this happens you may wish to use `setf` to modify the value of the special variable

```
*search-limit*
```

which sets an upper limit on the number of incomplete plan states that UCPOP explores when solving a problem. This allows UCPOP to avoid swamping a machine when it is given a very hard, or unsolvable, problem.

Since the function `bf-control` performs a best first search, it uses exponential space. Another way to make UCPOP avoid swamping a machine is to use the function `ibf-control` instead. This function performs an iterative deepening best first search as defined in [**?**].

3.4 Graphic Search-Space Browser

The file `vcr.lisp` contains an X-Window interface for analyzing a search tree created by UCPOP. It is invoked using the routine `bf-show` which takes a problem as first argument (as does `bf-control`). An example invocation appears below.

```
(bf-show 'sussman-anomaly "mizar:0")
```

The first parameter specifies a problem, and the second specifies an X-Window display. The routine first executes the planner on the problem, and then raises a window for displaying the search tree. The window is initially empty. The search tree is brought up by clicking on a "refresh" button. The meanings of all the buttons are:

- <<: Backup the VCR until a mouse button is pressed.
- <: Backup the VCR to the plan visited just prior to the current one.
- >: Progress the VCR to the next visited plan.
- >>: Progress the VCR until a mouse button is pressed.
- **Resize**: Shrink or enlarge the displayed search tree by a constant.
- **Refresh**: Format and refresh the search tree display.
- **View**: Change the node picture of the tree nodes to one of the following:
  - **Order**: Color the nodes from red to purple depending on visitation order.
  - **Histograms**: Display the standard histogram icon.
  - **Rank**: Color the nodes from red to purple depending on the absolute rank.
- **Summary**: Select one of the following tree summarization commands:
  - **All Nodes**: Display all nodes in the tree.
  - **Steps Only**: Display nodes for plans with newly added steps.
  - **Solution Only**: Display solution node only (if found).
  - **Solution Path Only**: Display nodes on solution path (if found).
  - **Expand Node**: Display the children of the current plan.
  - **Contract Node**: Do not display the children of the current plan.
  - **Expand Branch**: Display the branch rooted at the current plan.
  - **Contract Branch**: Do not display the branch rooted at the current plan.
- **Exit**: Close the window and exit the `bf-show` routine

Each node in the search tree graph represents a partial plan and the path to the solution plan is left justified. The plans are graphically summarized with four histograms. From left to right these histograms represent the number of unsolved goals, causal links, threats, and steps in a plan. A very brief summary of the reason for creating a plan appears as a label below the histograms, when there is enough space. For example, the label "5:PUTON" means that the fifth step was created, and it was a PUTON action.

At any given moment, one of the plans in the tree is the *current plan*. This plan is marked by a black box, and its histograms appear in the lower left corner. The *current plan* can be changed by the progress/backup buttons or by clicking on another plan in the tree.

Clicking on the current plan displays it textually in the lower right corner. A more verbose reason for creating the plan appears after the plan.

Since UCPOP creates more plans than it visits, some of the VCR's tree nodes were never visited. It is possible to manually visit such a node by clicking on a plan flaw. These flaws appear in the plan's textual display in the lower right corner. It is possible for the VCR to incorrectly format the tree while manually visiting plans. A "refresh" will reformat and redisplay the tree.

When a search tree is initially displayed, only the plans on a path to the solution appear with their graphical summarization. The other plans are graphically summarized once they are made current. It is possible to make plans appear without ever making their parents appear. This happens when using the progress or backup buttons after clicking on a plan in the middle of the search path.

4 Creating New Domains

Planning domains are defined using an action description language that is inspired by KRSL [?] but corresponds more closely to ADL [?] in expressive content. This section describes the salient features of UCPOP's language and provides its grammar.

4.1 Operators

The EBNF[4] for the operator definition is:

```
<operator> ::= (define (operator <operator name>)
                 :parameters   (<parameter>*)
                [:precondition <GD>]
                 :effect       <effect>)

<parameter> ::= <variable-name>
<parameter> ::= <typed-var>
```

The *parameters* list is simply the list of variables on which the particular rule operates, *i.e.*, its arguments. The *precondition* is an optional goal description (GD) which must be satisfied before the operator is applied. As defined below, UCPOP goal descriptions are quite expressive: an arbitrary function-free first-order logical sentence is allowed. If no preconditions are specified, then the operator is always applicable. *Effects* list the changes which the operator imposes on the current state of the world. Effects may be universally quantified and conditional, but full first order sentences (*e.g.*, disjunction and Skolem functions) are not allowed. Thus, it is important to realize that UCPOP is asymmetric: action preconditions are considerably more expressive than action effects.

Free variables are not allowed. All variables in an operator definition (*i.e.*, in its preconditions or effects) must be included in the parameter list or explicitly introduced with a quantifier.

[4]Our EBNF definitions use square brackets ([and] to surround an optional clause. We use an asterix (*) to mean "zero or more of". Angle brackets denote names. Ordinary parenthesis are an essential part of the syntax we are defining and have no semantics in the EBNF meta language.

4.2 Goal Descriptions

A goal description is used to specify the desired goals in a planning problem and also the preconditions for an operator. Function free first order predicate logic (including nested quantifiers) is allowed.

```
<GD> ::= <term>
<GD> ::= (and <GD>*)
<GD> ::= (or  <GD>*)
<GD> ::= (not <GD>)
<GD> ::= (imply <GD> <GD>)
<GD> ::= (forall <term> [ <GD> ] )
<GD> ::= (exists <term> [ <GD> ] )
<GD> ::= (eq <argument> <argument>)
<GD> ::= (neq <argument> <argument>)
```

where `eq` and `neq` specify equality and inequality constraints between `<argument>`s respectively. In order to facilitate the definition of domains from outside the `"UCPOP"` package, all of the keywords mentioned above can also be prefixed with a ":". Finally, a `<term>` is an atomic expression of the form:

```
<term>     ::= (<predicate-name> <argument>*)
<argument> ::= <constant-name>
<argument> ::= <variable-name>
```

For example, one can create a goal requiring that all blocks be clear with `(forall (block ?b) (clear ?b))`. When confronted with a goal of this form, UCPOP takes the initial conditions and creates a goal `(clear` *B*`)` for every block *B*. This approach works for static predicates which cannot appear in any operator's effects.

When a predicate does appear in one or more operators' effects, it becomes *dynamic*. For example, the existence of an operator that creates or destroys a block would make `block` a dynamic predicate. In this event, UCPOP adds the goal `(or (not (block` *B*`)) (and (block` *B*`) (clear` *B*`))` for every block *B*. Thus, each block must either be cleared or deleted in order to achieve the goal.

Since steps can add dynamic predicates, UCPOP must consider all steps as well as the initial conditions when handling `forall` goals. In the previous example, each step with an effect `(block` *B*`)` makes UCPOP create a disjunctive goal like the one previously mentioned. In order to maintain correctness, UCPOP maintains a list of previously handled `forall` goals. These goals cause the creation of disjunctive goals whenever a new step with an effect, like `(block` *B*`)`, is inserted into a plan [5].

In a more complex example, the first argument of a `forall` equation can be an arbitrary term. For example, one can create a goal requiring that all blocks on the table are clear with `(forall (on ?b Table) (clear ?b))`. When confronted with a goal of this form, UCPOP detects effects like `(on ?x ?y)` and generates disjunctive goals like:

[5] This technique requires one restriction on domain encodings. When a step adds `(block` *B*`)`, *B* cannot be a universally quantified variable. In general, any `<term>` in a `forall` or `exists` goal equation cannot be positively asserted by a universal effect. UCPOP actively enforces this restriction.

```
(or (neq ?y Table)
    (and (eq ?y Table)
         (or (not (on ?x ?y))
             (and (on ?x ?y)
                  (clear ?x)))))
```

In this case the `forall` goal can be satisfied three different ways. In the first two the effect does not affect the `forall` goal because `(neq ?y Table)` is true, or a step asserting `(not (on ?x ?y))` intervened. If `(on ?x ?y)` is true at the time of the `forall`, then `(clear ?x)` must also be true. Considering all three options ensures completeness. The syntax and semantics of `forall` and `exists` was inspired by PRODIGY[?].

Finally, the `<GD>` is optional in `forall` and `exists` equations. This lets the user specify goals like `(not (exists (on ?x b)))` to require that a block `b` is clear.

4.3 Effects

UCPOP allows both conditional and universally quantified effects. The description is straightforward:

```
<effect> ::= (and <effect>*)
<effect> ::= (forall (<variable-name>*) <effect>)
<effect> ::= (when <GD> <effect>)
<effect> ::= (not <term>)
<effect> ::= <term>
```

This definition of `<effect>` is less verbose than that of UCPOP version 1.00a. The user has the option defining operators using this formalism, or the old formalism.

As in STRIPS, the truth value of predicates are assumed to persist forward in time. Unlike STRIPS, UCPOP has no delete list — instead of deleting `(on a b)` one simply asserts `(not (on a b))`. If an operator's effects does not mention a predicate P then the truth of that predicate is assumed unchanged by an instance of the operator. The initial conditions, however, while represented as the effects of a dummy "start" step, are treated differently. All predicates which are not explicitly said to be true in the initial conditions are assumed by UCPOP to be false.

4.4 Axioms

Domain axioms provide a convenient way to structure domains so that action effects can be short and sweet. For example, suppose that one wished to represent both `above` and `on` predicates in the blocks world. Using the STRIPS representation (or even ADL) one would need to explicitly show how each action affected *both* predicates.

This leads to poor software engineering (or perhaps we should say "domain engineering"). For example, if one added a new predicate (perhaps `below`) then one would have to go and change *every* action definition. To avoid this error prone problem, version 2.0 provides a restricted form of domain axioms. We restrict the form of axioms to keep the frame problem in check.

The basic idea is for the user to partition the domain predicates into two sets: primitive and derived. For example, `on` might be primitive and `above` might be derived. Actions can only affect primitive terms while axioms are restricted to assert derived predicates by defining them in terms of a `<GD>` which may include both primitive and derived terms. UCPOP actively enforces this partition.

```
<axiom> ::= (define (axiom <axiom name>)
              :context <GD>
              :implies <term>)
```

For example, we might define `above` as follows:

```
(define (axiom is-above)
  :context (or (on ?x ?y)
               (and (exists (on ?x ?z) (above ?z ?y))))
  :implies (above ?x ?y))
```

As another blocks world example, we could write an axiom for defining the derived term `(clear ?x)` as follows:

```
(define (axiom is-clear)
  :context (or (eq ?x Table)
               (not (exists (on ?b ?x))))
  :implies (clear ?x))
```

UCPOP handles a goal with a derived term by nondeterministicly choosing an appropriate axiom and replacing the goal with a new goal containing the axiom's `:context`. Since a `<GD>` can contain derived terms, an infinite recurse is possible. UCPOP does not detect such a recurse.

4.5 Facts

Facts are preconditions that are satisfied by calling lisp functions. They are typically used for defining complex relationships, such as arithmetic functions and the definition of new constant symbols. Each fact predicate has a unique lisp function associated with it. A `define` macro is used to create facts and associate lisp functions with them.

```
<fact> ::= (define (fact (<predicate-name> <variable-name>*))
             <function-body>)
```

One simple example of a fact, the arithmetic $<$ operation, appears below and is used as a filter when planning.

```
(define (fact (less-than ?x ?y))
   (cond ((or (variable:variable? ?x) (variable:variable? ?y))
          :no-match-attempted)
         ((and (numberp ?x) (numberp ?x) (< ?x ?y))
          '(nil))
         (t nil)))
```

The calling conventions for these functions are similar to those in PRODIGY[?]. When resolving a fact goal, UCPOP evaluates the associated lisp function within the context of the goal term. For instance, the goal `(less-than 5 13` would make UCPOP evaluate the function with `?x` and `?y` being bound to the symbols 5 and 13 respectively. This function either evaluates to `:no-match-attempted`, `nil`, or `'(nil)`. The `:no-match-attempted` symbol is used to inform UCPOP that there is not enough information to evaluate this fact. When such is the case, the associated goal is differed until later. The lists `'(nil)` and `nil` inform UCPOP that the fact is satisfied and unsatisfyable respectively. The reasons for these lists will become apparent by the end of this section.

The next example fact comes from the `office-world` domain in `domains.lisp` file. It is used to define new constant symbols in a dynamic universe. In this example a fact returns a list of binding constraint lists.

```
(define (fact (new-object ?x))
    (when (variable:variable? ?x)
      (list (setb ?x (gensym "obj-")))))
```

A binding constraint list is an association list where each entry specifies an equality constraint. The fact goal is resolved by adding these constraints to the plan. Since there can be more than one set of binding constraints to resolve a fact goal, the fact function must evaluate to a list of one or more binding constraint lists. UCPOP nondeterministicly chooses one of these constraint lists to resolve the goal. Finally, note that the list of constraint lists `(nil)` specifies that the fact is resolved without adding any variable constraints, and `nil` specifies that the fact cannot be resolved at all.

5 Creating a Search Controller

UCPOP solves a planning problem by taking an initial dummy plan, and performing a best first search through a space of plans for a solution. When UCPOP visits a plan it expands the search space by choosing a flaw with that plan (e.g. a threat) and modifying the plan to fix that flaw (e.g. promotion). The search controller uses a *production system* to guide this process. Such an approach to search control is by no means new. Our control language was influenced by search control in PRODIGY[?].

To run a simple example that uses a controller defined in `controllers.lisp` and a problem from `domains.lisp` you can type:

```
(sc-control 'uget-paid #'bf-mimic)
```

This defines and runs the example problem mentioned earlier. The search controller created by the function `bf-mimic` makes `sc-control` perform a search identical to the one performed by the `bf-control` function.

The routine `sc-show` also performs a controlled search. After the search it functions just like `bf-show` in that it raises a window on an X-Window display for analyzing the tree. To call this function you can type:

```
(sc-show 'sussman-anomaly #'bf-mimic "mizar:0")
```

The parameters, from left to right, specify the problem, controller, and X-Window display.

Finally, the function `isc-control` performs a controlled search using iterative deepening. With `bf-mimic`, `isc-control` will mimic `ibf-control`. In general, any controller that works with `sc-control` will also work with `isc-control`.

5.1 Search Control Rules

The search control rule language is a lot like prolog, but it works in a forward chaining direction whereas prolog is backward chaining.

When writing search control rules it is important to keep in mind that there are two types of clauses, triggering and filtering. When a new proposition is stored in the database, a rule with triggering clauses that match will get activated. If the rule's filtering clauses are satisfied with the bindings produced by the triggering clauses, then the rule fires and its consequent takes effect. Note that it doesn't make sense to write an scr rule that doesn't have at least one triggering clause in its antecedant, since it would never do anything.

You can define more filtering clauses with the `(define (clause ...))` macro, and more triggering clauses in the effects of rules. The order of the clauses in the antecedant of your search control rules is important - you should put the triggering clauses first.

The syntax of a search control rule definition is:

⟨*scr*⟩ ::= `(define (scr` ⟨*rule-name*⟩`)`
`:when '(`⟨*pattern*⟩ ⟨*pattern*⟩∗`)`
`:effect '`⟨*pattern*⟩`)`

⟨*pattern*⟩ ::= `(`⟨*predicate*⟩ ⟨*argument*⟩∗`)`

A ⟨*predicate*⟩ must be a symbol, but an ⟨*argument*⟩ can be any arbitrary symbolic expression where symbols begining with `$` represent variables. For instance the pattern `(test $f)` matches clauses `(test` n`)` for any value of n.

5.2 Triggering Clauses Asserted by UCPOP

When UCPOP removes a plan from the priority queue and visits it (i.e., starts to work on it), it asserts the following clauses into the database:

`(:current :node` ⟨*plan*⟩`)`	The current ⟨*plan*⟩ that UCPOP is visiting.
`(:flaw` ⟨*flaw*⟩`)`	Each of the newly introduced flaws in the current plan generates an entry of this form. (The older flaws are still on the priority queue of flaws for this plan)

At this point any search control rule that has an antecedant pattern matching one of these clauses will wake up and try to fire. Once all rules stop firing, for each `:flaw` clause a set of `:rank` clauses are combined to compute a `:candidate` clause. These computed clauses are added to the database and look like the following:

`(:candidate :flaw` ⟨*rank*⟩ ⟨*flaw*⟩`)`	The ranks of the new flaws in the current plan. Flaws with low numbered ranks will be repaired first.

The value of ⟨*rank*⟩ for some flaw F is the sum of all the R values in (`:rank :flaw` R F) clauses asserted during the previous rule firings. The default rank is zero when there were no assertions. These `:candidate` assertions may cause more rules (i.e., preference rules for flaws) to be triggered; once these rules stop firing, the search controller chooses the most interesting flaw to repair and generates all possible refinement plans that fix the flaw. The following clauses are asserted at this point.

(`:current :flaw` ⟨*flaw*⟩)	The ⟨*flaw*⟩ that UCPOP is repairing while visiting the current plan.
(`:node` ⟨*plan*⟩ ⟨*reason*⟩)	The plans created when repairing the current flaw in the current plan. The format of ⟨*reason*⟩ is one of the following patterns:
⟨*reason*⟩ ::= (`:init`)	- The start plan
(`:fact` ⟨*term*⟩)	- Fact ⟨*term*⟩ was handled
(`:goal` ⟨*term*⟩ ⟨*s*⟩)	- Precondition ⟨*term*⟩ was added for step ⟨*s*⟩
(`:step` ⟨*s*⟩ ⟨*term*⟩)	- Step ⟨*s*⟩ was added to assert ⟨*term*⟩
(`:link` ⟨*s*⟩ ⟨*term*⟩)	- Step ⟨*s*⟩ was reused to assert ⟨*term*⟩
(`:cw-assumption`)	- Made a closed world assumption (for :not terms)
(`:bogus`)	- This flaw did not really exist
(`:order` ⟨s_1⟩ ⟨s_2⟩)	- order step ⟨s_1⟩ before step ⟨s_2⟩

These assertions will trigger more rules. Once these rules stop firing, `:rank`, `:select`, and `:reject` clauses are collected to define the *candidate* plans that will get placed into UCPOPś priority queue for futher consideration. For each such plan a clause of the following form is asserted:

(`:candidate :node` ⟨*rank*⟩ ⟨*plan*⟩)	The ranks of the `:node` (i.e. plan) entries. One of these will get added to the database for each nonrejected plan unless *some* plan was selected. If any plan is selected, then one of these assertions will be added for each nonrejected plan that was selected.

Once the rules that these assertions trigger stop firing, the candidate plans are sorted by their rank values followed by preferences defined by asserted `:prefer` clauses. At this point the plans get placed into the priority queue, the database is flushed, and UCPOP removes the next plan from the priority queue.

5.3 Rule Consequents

A rule can assert any clause, but only a limited number directly affect UCPOPś behavior. They are:

`(:rank :node` ⟨*rank*⟩ ⟨*plan*⟩`)`	ranks a plan with the associated value. Low numbered plans are considered before higher numbered ones. The default rank is 0. A plan's final rank is the sum of all rules that assert rankings.
`(:rank :flaw` ⟨*rank*⟩ ⟨*flaw*⟩`)`	similar
`(:reject :node` ⟨*plan*⟩`)`	rejects a plan by removing its ranking.
`(:select :node` ⟨*plan*⟩`)`	specifies that all plans that are not selected get rejected. Selection takes precedence over rejection.
`(:select :flaw` ⟨*flaw*⟩`)`	similar
`(:prefer :node` ⟨p_1⟩ ⟨p_2⟩`)`	As long as ⟨p_1⟩ and ⟨p_2⟩ are of the same rank, then put ⟨p_1⟩ ahead of ⟨p_2⟩ on the priority queue (i.e., visit and repair it first).
`(:prefer :flaw` ⟨f_1⟩ ⟨f_2⟩`)`	similar

5.4 Predefined Filtering Clauses

You can define your own filtering clauses with the `(define (clause ...))` syntax, but this usually requires understanding the UCPOP internal datastructures. To simplify your life, we provide some predefined clauses.

`(operator` ⟨*s*⟩ ⟨*action*⟩ ⟨*plan*⟩`)`

This clause is true for each step ⟨*s*⟩ in plan ⟨*plan*⟩ that uses operator ⟨*action*⟩. For example, suppose you want your scr rule to only trigger on plans whose 5th step (note 5 denotes the fifth step added to the plan, not the fifth step to eventually be executed!) is a `(puton $a $b)` action. In this case you could add the following filtering clause: `(operator 5 (puton $a $b) $p)`. Presumably, `$p` would be bound by a previous triggering clause, perhaps `(:node` ⟨*plan*⟩ ⟨*reason*⟩`)`. Note that variables `$a` and `$b` get bound to the operands of the puton so you can test them with subsequent clauses. Note further, that it's handy to know what ⟨*s*⟩ is because it lets you focus on the step that was most recently added to the plan.

`(goal` ⟨*plan*⟩ ⟨*flaw*⟩ ⟨*term*⟩ ⟨*step*⟩`)`

This clause is true for each ⟨*flaw*⟩ in ⟨*plan*⟩ that is an unsupported precondition ⟨*term*⟩ for step number ⟨*step*⟩. Naturally, this is useful when ranking flaws.

(threat ⟨*plan*⟩ ⟨*flaw*⟩ ⟨*link*⟩ ⟨*s*⟩)

This clause is true for each ⟨*flaw*⟩ in ⟨*plan*⟩ that is states that causal link ⟨*link*⟩ is

threatened by step ⟨*s*⟩. Naturally, this is useful when ranking flaws.

`(neq` ⟨*x*⟩ ⟨*y*⟩`)`
This clause is true in all cases where ⟨*x*⟩ is not lisp EQ to ⟨*y*⟩.

5.5 Useful Functions For Defining Rules

The following lisp function isn't a clause or an scr rule, but it *generates* an scr rule.

`(fail-link` ⟨*producer*⟩ ⟨*term*⟩ ⟨*consumer*⟩`)`
This routine produces rules that reject plans that add a step with action ⟨*producer*⟩ to add effect ⟨*term*⟩ for a step with action ⟨*consumer*⟩. For example:

```
(fail-link '(close $a) '(:not (open $a)) '(open $a))
```

creates a rule that ensures that the planner never closes a container in order to open it again.

It's instructive to look at the rule that this invocation creates. (See the source code for fail-link in controllers.lisp). The resulting scr has the following antecedant:

```
:when `((:current :node $n)
        (:current :flaw $f)
        (:node $p (:step $s2 (:not (open $a))))
        (operator $s2 (close $a) $p)
        (goal $n $f $g $s1)
        (operator $s1 (open $a) $p))
```

Note that the first three clauses are triggering clauses. This rule will wake up when plan `$p` has been generated as a refinement of plan `$n` in an attempt to fix flaw `$f` by adding a new step (number `$s2`) that produces the `(:not (open $a))` proposition. The rule's effect will be enforced when the new step (just added) is a `(close $a)` action, and the current flaw (ie the one repaired by refining plan `$n` to plan `$p`) is an open (unsupported) goal of step `$s1`, and `$s1` is an `(open $a)` action. (Note that the variable `$g` is just a dummy placeholder - it's value never gets tested or used). When these conditions are all satisfied, the rule will fire and

```
:effect '(:reject :node $p)))
```

will ensure that `$p` will never get put onto the priority queue, so it will never get visited.

5.6 Defining Filtering Clauses

A rule's precondition contains patterns that match two types of clauses: triggering clauses and filtering clauses. Where a triggering clause uses the database to specify a simple relationship, a filtering clause uses a lisp to specify a complex relationship. The syntax of a filtering clause definition is:

⟨*clause*⟩ ::= `(define (clause (`⟨*predicate-name*⟩ ⟨*variable-name*⟩`*))`
⟨*function-body*⟩`)`

For example, consider the following rule. The first pattern matches a trigger clause. When the rule is triggered, the values for `$p` and `$r` are found, and the search controller queries the `rank-plan` relationship with these values.

```
(define (scr select-ranked)
    :when '((:node $p $r) (rank-plan $p $n))
    :effect '(:rank :node $n $p))
```

The `rank-plan` relationship associates plans with some other value ⟨*rank*⟩. We define define the relationship it with the following lisp function:

```
(define (clause (rank-plan plan rank))
  (when (plan-p plan)
    (matchb `(rank-plan ,plan ,rank)
            `((rank-plan ,plan ,(rank3 plan))))))
```

Whenever a the search controller queries with a `(rank-plan` ⟨*plan*⟩ ⟨*rank*⟩`)` pattern, the function body of the above definition is called with `plan` and `rank` bound to the ⟨*plan*⟩ and ⟨*rank*⟩ arguments respectively. If `plan-p` accepts ⟨*plan*⟩, the function will compute a list of relevant clauses and invoke `matchb` to match the pattern against this list. Only relevant clauses that match the pattern are returned to the search controller.

Actually, `matchb` does not return a list of clauses. It returns lists of variable binding lists. Each variable binding list corresponds to the results of a match between the pattern and a clause that would have been returned. This observation can be used to optimize a filtering clause. For example, the above clause can be replaced with:

```
(define (clause (rank-plan plan rank))
  (when (plan-p plan)
    (matchb rank `(,(rank3 plan)))))
```

5.7 Example Controller: `bf-mimic`

The routines `sc-control` and `sc-show` call a user specified function to define the controller. This function takes a problem description as its only argument. This allows the creation of problem specific controllers. Since the controller defined by the function `bf-mimic` is problem independent, the first argument is ignored here.

```
(defun bf-mimic (prob)
  (declare (ignore prob))
  (reset-controller)
  (define (scr select-ranked)
      :when '((:node $p $r) (rank-plan $p $n))
      :effect '(:rank :node $n $p))
  (define (scr select-threats)
      :when '((:current :node $n)
```

```
            (:flaw $g1)
            (threat $n $g1 $l $t))
        :effect '(:rank :flaw -1 $g1))
    (define (clause (rank-plan p n))
        (bound! 'rank-plan p)
      (when (plan-p p)
        (matchb n (list (rank3 p))))))
```

This function starts by initializing the UCPOP controller using the function `reset-controller`. At this point the controller has no rules in its production system. When a search control rule is defined, it is immediately inserted into the controller.

The purpose of the rule `select-ranked` is to rank plans prior to their insertion into UCPOPś priority queue. It ultimately controls the order in which UCPOP visits partial plans in its search space. Similarly the `select-threats` rule directs UCPOP to repair unsafety conditions before considering open conditions. For more examples of rule definitions, see the file `controllers.lisp`.

Finally, `bf-mimic` defines a filtering clause to assist the `select-ranked` rule in computing a ranking value for a partial plan. This value is computed by the function `rank3` which is the domain independent ranking function used by `bf-control`. This filtering clause is functionally identical to the one mentioned previously, but it uses `bound!` to test for errors. The function `bound!` is similar to a lisp `assert`. If any of its arguments are variablized patterns, then it stops all planning and flags an error.

5.8 Debugging a Search Controller

In order to facilitate debugging a search controller there exist the routines `trace-scr` and `untrace-scr`. They are used to trace the use of specific rules and clauses, and untrace respectively. Example invocations appear below.

```
(trace-scr 'select-threats)
(untrace-scr)
```

Another useful pair of routines are `profile` and `show-profile`. The first turns on production profiling, and the second display profiles and turns production profiling off. A profile displays how often each rule was triggered and how often it fired. The invocations are:

```
(profile)
(show-profile)
```

6 Primitive UCPOP Calls

While casual users will be happy to use the problem-oriented interface to UCPOP which was described in section 3, many users will desire more control over the planner. This section shows how.

6.1 Switches

■ is this implemented? `*ord-constrain-on-confront*` with default value of NIL. When true, DISABLE introduces ordering constraints $S_{prod} < S_{threat} < S_{consume}$ when the threat is resolved via confrontation.

■ is this implemented? `*positive-threats*` (default NIL). When true, causes ucpop to recognize a step as a threat when the step provides redudant causal support for a consumer. When the switch is NIL then steps are threats only when the effect unfies with the negation of the link's label. ■

There are several global variables used to control UCPOP. For instance, the variable `*search-limit*` was already mentioned in section 3.3. Some other important variables are `*templates*`, `*axioms*`, and `*facts*` which contain the operators, axioms, and facts of the domain. These variables are reset with the function `reset-domain` which takes no arguments. Each call to `define` with type arguments `operator`, `axiom`, or `fact` adds an entry to one of these variables.

The variable `*verbose*` is used to control how plan and stat objects are printed. Set it to `T` and the objects will be displayed in all their glory. By default it is set to `nil` and the routines `display-plan` and `display-stat` can be used to see the details of the objects which are especially interesting.

The variable `*d-sep*` is used to control the definition of an unsafety condition. When it is set to `nil`, all steps that can affect a link are reported as unsafety conditions. Otherwise, unsafety conditions only occur when a step affects a link without the addition of binding constraints. See [?] for a more detailed explanation of this strategy.

6.2 Calling the Planner

The basic call to UCPOP is through the function `plan` which takes two required and two optional keyword arguments:

- `initial` This is a list of terms describing the initial conditions of the planning problem. Both true and false (*i.e.*, `(not ...` terms are ok, but since all predicates are assumed initially false unless explicitly stated to be true, the use of `not` is unnecessary. `domains.lisp` contains many examples using `not`, but this is for historical reasons.

- `goals` This is the GD (goal description) for the planning problem.

- `rank-fun` This optional keyword argument is a function which must take a single argument (a plan structure) and returns a number that indicates how good the plan is. Low numbers denote good plans. The default value is `#'rank3` which returns the sums of the number of plan steps, the number of open conditions, and the number of threatened links.

- `search-fun` This optional keyword argument is a search control function (default value `#'bestf-search`) which is expected to take five arguments: an initial state, a successors function, a goal test function, a ranking function and a limit on the states it will explore.

The search function should return three values, the best plan found, and two statistics: the average branching factor and the number of generated but unexplored states.

`plan` returns two values, the best plan found and a statistics object which has more information than most people desire to look at.

6.3 Bugs and Versions

If you ever find a bug or have a suggestion regarding UCPOP please send electronic mail to internet address `bug-ucpop@cs.washington.edu`. Please be sure to include the software version you are running which can be determined by evaluating the symbol `*version*`. In addition, please describe the type of machine you are running on and the specific implementation of COMMON LISP (also with version) being used. If possible, include a trace of the bug in action. We aim to continue improving the UCPOP code.

7 UCPOP Internals

In this section we describe the main UCPOP data structures which are used to represent plans, individual actions (*i.e.*, plan steps), action effects, causal links, open preconditions, and threatened (*i.e.*, "unsafe") links.

7.1 Plans

The main data structure in UCPOP is a defstruct called `plan` which has the following fields:

- `steps` This is a list of `p-step` structures which encode the basic actions in the plan.
- `links` A list of causal `link` structures.
- `flaws` A list of `unsafe` and `openc` structures denoting threatened links and unsupported step preconditions.
- `ordering` A list of lists. Each sublist is of the form `(ID1 ID2)` where the two identifiers refer to steps in the plan and the first one is being constrained to precede the second.
- `foralls` A list of `forall` structures. Each structure represents a previously processed universal goal and has to be checked whenever a new step is added to a plan.
- `bindings` The binding constraints are represented as a list of variable codesignation sets.
- `high-step` This is an integer denoting the identifier of the step last added.
- `other` This is an association list for search control and debugging purposes. You might wish to put stuff here as well.

7.2 Plan Steps

Plan steps are represented as `p-step` structures with the following fields:

- `ID` An integer step number.
- `action` The name of the action — a formula such as `(puton ?X ?Y)` .
- `parms` The parameters for the step — a list of variables.
- `precond` Preconditions like `(clear ?X)`.
- `add` These are the `effects` asserted by the step.
- `cache` A cache of existing steps.

7.3 Effects

An `effect` is represented with a structure with four fields:

- `ID` An integer denoting the identifier of the step that owns this effect.
- `forall` A list of variables that are universal in the effect.
- `precond` The preconditions of the effect (a GD, *i.e.*, a goal description).
- `add` A list of terms that the effect adds.

7.4 Causal Links

Each `link` structure has three fields:

- `id1` The identifier of the producing step.
- `condition` The term being supported.
- `id2` The identifier of the consuming step.

7.5 Open Conditions

Preconditions which have yet to be supported with a causal link are represented with `openc` structures, each of which have two fields:

- `condition` The open precondition itself.
- `id` Identifier of the step whose precondition needs to be supported.

7.6 Threatened Links

Causal links which are threatened by a third step are represented with structures called `unsafes`. These structures have three fields:

- `link` An identifier for the `link` being threatened.
- `clobber-effect` The `effect` which is doing the threatening.
- `clobber-condition` The added condition (part of the effect) which causes the threat.

7.7 Forall Goals

The equation `(forall (block ?x) (clear ?x))` forms an example of a universal precondition. `(block ?x)` is called the *generator* and `(clear ?x)` is the *condition*. UCPOP maintains completeness by recording such preconditions in `forall` structures, each of which has four fields:

- `ID` The identifier of the step with a universal precondition.
- `vars` A list of the universal variables in the goal.
- `type` The generator of the universal precondition.
- `condition` The goal required for each set of variable binding constraints that makes the generator true.