UCPOP User's Manual

(Version 4.0)

Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden,
Scott Penberthy, Ying Sun, & Daniel Weld

Technical Report 93-09-06d
November 6, 1995

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 Grants IRI-8957302 and IRI-9303461, Office of Naval Research Grants 90-J-1904 and N00014-94-1-0060, Apple Computer, Rockwell International, and the Xerox corporation.

Abstract

The UCPOP partial order planning algorithm handles a subset of the KRSL action representation [1] that corresponds roughly with Pednault's ADL language [12] and PRODIGY's PDL language [10]. 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. Features include:

- Universal quantification over dynamic universes (*i.e.*, object creation and destruction),
- Domain axioms over stratified theories,
- "Facts" *i.e.* predicates expanding to lisp code,
- Specification of safety constraints in domain definition,
- Advanced CLIM-based graphic plan-space browser with multiple views,
- Declarative specification of search control rules,
- Large set of domain theories & search functions for testing,
- Expanded users manual and tutorial introductions.

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 [12], including actions with conditional effects, universally quantified preconditions and effects, and universally quantified goals. UCPOP has desirable formal properties: *e.g.*, [13] 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. Confrontation: if the threat results from a conditional effect, add a new subgoal that negates the preconditions of the offending effect.[3]

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 `doc/kr92-ucpop.ps` which contains a postscript version of the original paper [13]. For a tutorial introduction to least commitment planning (with several comprehensive examples) see the file `doc/planning-intro.ps` [16].

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

[3] Setting the `*dsep*` switch to `nil` allows UCPOP to use *separation* as an additional threat protection mechanism. Separation add binding constraints (when consistent) that ensure that S_k cannot interfere with r. However, as explained in [14] separation is unnecessary and unproductive if threat processing is delayed.

1.2 New Features for Version 4.0

Many new features have been added to UCPOP. The plan-space browser, originally known as the VCR, has been renamed PDB, or Plan DeBugger. Like previous versions, it supports multiple views of plans, comparison of sibling plan structures, and search control debugging, but now includes tree comparison, domain browsing and an improved user interface.

The domain definition language has been enhanced to improve quantification and handle dynamic universes — action effects can create new objects or delete existing ones and quantified goals will work properly. Another addition is safety constraints, background goals that UCPOP would check and ensure no violation before adding any new action into a plan. Section 4.7 describes how users can specify safety constraints in a domain. The language is also extended to allow specification of primary and side effects of actions to control search.

UCPOP supports ZLIFO, an alternative flaw ranking function which makes the planner go faster on many problems [15]. Type `(turn-zlifo-on)` and `(turn-zlifo-off)` to toggle all the ZLIFO settings at once. ZLIFO does not work on domains with universally quantified variables. See the file `zlifo.doc` for more details.

1.3 New Features in Previous Versions

UCPOP supports domain axioms as explained in section 4.5. 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 4.6 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.

UCPOP continues to support the search control facility. It 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. Moreover in addition to ranking functions for plans, the planner now provide hook functions to rank flaws as well.

Note that the operator and domain definition syntax is different (improved to be more regular), but all old domain definitions will still work; see section 4 for the details.

1.4 Document Overview

Section 2 explains installation and loading. Section 3 describes two ways to run the UCPOP planner: from the lisp read-eval-print loop and via the PDB graphical interface. Section 4 describes the UCPOP action language in detail and explains how to define new domains. Section 5 describes UCPOP's search control facility, while section 6 shows several methods for debugging search control. Appendix A lists some of the primitive calls and switches that

UCPOP supports. Finally, Appendix B 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
`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
`safety.lisp`	Safety constraints
`domains.lisp`	Example domain and problem definitions
`controllers.lisp`	Example search controller definitions for UCPOP
`zlifo.lisp`	Enhancements to the planning algorithm

An optional CLIM graphical interface module is also available, and consists of the following files:

`pdb/compute-tree`	Tree manipulations.
`pdb/dialogs`	Dialogs.
`pdb/drawing`	Support routines for drawing.
`pdb/layout`	Formatting routines for graphics.
`pdb/recording`	Recording planning sessions.
`pdb/structs`	Class definitions.
`pdb/translation`	Correspondences between structures and graphics.
`pdb/window-buttons`	Support macros.
`pdb/shell`	Basic application structure.
`pdb/shell-commands`	Plan overview window commands.
`pdb/shell-present`	Plan overview window graphics.
`pdb/vcr-commands`	VCR window commands.
`pdb/vcr-present`	VCR window graphics.
`pdb/plan-commands`	Plan window commands.
`pdb/plan-present`	Plan window graphics.
`pdb/browser-commands`	Browser window commands.
`pdb/browser-present`	Browser window graphics.

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[4] you need only type:

[4] 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.

```
(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 (described in section 3.2) 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 PDB if appropriate.

3 Operating UCPOP

There are two ways to operate UCPOP. From the Lisp read-eval-print loop one can call the planner using a variety of interface functions. These are described in Section 3.1.

The best way to operate UCPOP is via the graphical user interface known as the "Plan DeBugger" or PDB for short. This interface is discussed in Section 3.2.[5]

3.1 Running UCPOP from the Command Line

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.

3.1.1 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 [11] 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 [13] 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)))))) )
```

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

[5] PDB requires that your Lisp provide CLIM support; if in doubt, contact your Lisp vendor.

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 4.3).

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.

If `bf-control` is too verbose for your needs, try `bf-test`.

3.1.2 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 either the name of a predefined domain, or a function which will initialize the domain operators when called.
- `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 A for details on writing new ranking functions.
- `flaw-fun` An optional flaw ranking function to guide search. If not specified, it defaults to the standard domain-independent ranking function. See section A 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.

3.1.3 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 [9].

3.2 Operating UCPOP from the Graphical User-Interface

This section describes the PLAN DEBUGGER (PDB), a graphical plan-space browser built to support the UCPOP family of planners. If your Lisp implementation supports CLIM, then this is definitely the best way to operate UCPOP.[6]

The primary function of PDB is to allow the user to schedule planning problems to be run by the planner, record the actions taken by the planner, and then reconstruct that information in a meaningful way. Since UCPOP does extensive search in plan-space, PDB focuses on recording and presenting plan-space search trees generated by the planner. But PDB can also be used in a causal manner by users interested in exploring UCPOP behavior.

PDB is an interactive application which makes the UCPOP planning process more accessible to the user by the use of graphical interfaces to the planning process. It is intended to be used in several ways:

- Planner Performance Testing
- Domain Construction
- Search Control Development
- Planner Modification

PDB exists because of the difficulty involved in performing planner research without adequate visualization tools. In the case of UCPOP users who were trying to use the planner as well as programmers trying to improve and debug the system were having to wade through extensive trace information generated by the planner. PDB evolved as a way of presenting this information in a graphical format in order to make these tasks easier.

To get started with PDB load UCPOP and then type:

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

Now start PDB by typing:

```
(pdb-setup)
```

A window should appear and subsequent commands can be entered via menu manipulation.

A complete explanation of PDB operation is too lengthy for this manual — please refer instead to PDB's own reference manual, included as distribution file `doc/pdb.html`.

4 Creating New Domains

Planning domains are defined using an action description language that is inspired by KRSL [1] but corresponds more closely to ADL [12] in expressive content. This section describes the salient features of UCPOP's language and provides its grammar.

[6] If you do *not* run CLIM then you should seriously consider buying it (contact your Lisp vendor for details). CLIM will help you in many things besides UCPOP.

4.1 Domains

UCPOPrepresents domains either as a data structure containing all of the operators, axioms, and facts associated with that domain, or as a function which defines those structures when called. The EBNF[7] for defining a domain structure is:

```
<domain>        ::= (define (domain <domain name>)
                      [safety-def]
                      <structure-def>*)

<structure-def> ::= <operator-def>
<structure-def> ::= <axiom-def>
<structure-def> ::= <fact-def>

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

<axiom-def>     ::= (:axiom <axiom name>
                      :context <GD>
                      :implies <term>)

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

<safety-def>    ::= (:safety <name> <GD>)
```

4.2 Operators

The EBNF 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

[7] 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.

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.

4.3 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:[8]

```
<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

[8] Important note: UCPOP makes the following assumption about terms in its domain theory: it assumes that predicates have fixed arity. It is an extremely bad idea to use predicates with the same name but different number of arguments since optimization to the unify function may allow `(predicate A)` to unify with `(predicate ?x ?y)`.

`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 [9].

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:

```
(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[10].

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.

4.4 Effects

UCPOP allows both conditional and universally quantified effects. The description is straightforward:[10]

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

Note that universally quantified effects have a different syntax than universally quantified goals! In particular, universally quantified effects may introduce multiple variables,

[9] 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.

[10] 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.

but multiple, nested `forall` statements are necessary in preconditions and goals. Another difference is the restriction that the type be specified for variables in universally quantified preconditions.

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.

When `:side` is specified in front of an effect, the effect will be considered to be a side-effect. If `P` is a side effect of operator `O` then UCPOP will not add a new instance of `O` to the plan in service of an open `P` goal. However, the planner will link to an existing step in this situation. And an instance of such an operator will threaten a link labeled `(not P)`. Be careful with side effects since their use affects UCPOP completeness. Furthermore, the use of side effects elevates the importance of subgoal ordering. Without side effects, subgoal order does not affect UCPOP completeness, but with side effects it can.

Effect prefixed by `:primary` are not treated specially by the planner. They are used by the ranking function to give higher priority to "good" plans. Plans are more "good" if most links contain primary effects. See the function `rank4` in `domains/tire-ctlr.lisp` for an example of such a ranking function. The same file also contains `init-flat-tire` world with some side effects defined.

4.5 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, UCPOP 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.

4.6 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[10]. 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.

4.7 Safety Constraints

Safety constraints are background goals that must be satisfied throughout the planning process. Before adding any new action into a plan, UCPOP will check the safety constraints to see if there is any violation. If so, UCPOP will perform one of its repairs. The user needs to set the global variable `*safety-p*` to turn on this option.

```
<safety> ::= (define (safety <name>) <GD>)
```

For example, one could command a softbot (software robot) avoid deleting files that are not backed up on tape with the following constraint:

```
(define (safety file-safety)
  (:or (file ?f) (written-to-tape ?f)))
```

Free variables, such as $?f$ above, are interpreted as universally quantified.

It is important to note that safety constraints do *not* require an agent to *make* them true; rather, the agent must avoid creating *new* violations of the constraints. For example, if a constraint specifies that all of my files be read protected, then the agent would avoid any of my files to be readable; but if my `.plan` file is already readable in the initial state, then the agent would not protect that file.

For details of safety constraints, please refer to [17] which is included as distribution file `doc/first-law-aaai94.ps`.

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[10].

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:

$$\langle scr \rangle ::= \texttt{(define (scr } \langle \textit{rule-name} \rangle\texttt{)}$$
$$\texttt{:when '(}\langle pattern \rangle\ \langle pattern \rangle * \texttt{)}$$
$$\texttt{:effect '}\langle pattern \rangle\texttt{)}$$

$$\langle pattern \rangle ::= \texttt{(}\langle predicate \rangle\ \langle argument \rangle * \texttt{)}$$

A $\langle predicate \rangle$ must be a symbol, but an $\langle argument \rangle$ 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 $\langle plan \rangle$)	The current $\langle plan \rangle$ that UCPOP is visiting.
(:flaw $\langle flaw \rangle$)	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 $\langle rank \rangle$ $\langle flaw \rangle$)	The ranks of the new flaws in the current plan. Flaws with low numbered ranks will be repaired first.

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

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 UCPOP's priority queue for futher consideration. For each such plan a clause of the following form is asserted:

(:candidate :node $\langle rank \rangle$ $\langle plan \rangle$)	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 UCPOP's behavior. They are:

`(:rank :node` $\langle rank \rangle$ $\langle plan \rangle$`)`	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` $\langle flaw \rangle$ $\langle flaw \rangle$`)`	similar
`(:reject :node` $\langle plan \rangle$`)`	rejects a plan by removing its ranking.
`(:select :node` $\langle plan \rangle$`)`	specifies that all plans that are not selected get rejected. Selection takes precedence over rejection.
`(:select :flaw` $\langle flaw \rangle$`)`	similar
`(:prefer :node` $\langle p_1 \rangle$ $\langle p_2 \rangle$`)`	As long as $\langle p_1 \rangle$ and $\langle p_2 \rangle$ are of the same rank, then put $\langle p_1 \rangle$ ahead of $\langle p_2 \rangle$ on the priority queue (i.e., visit and repair it first).
`(:prefer :flaw` $\langle f_1 \rangle$ $\langle f_2 \rangle$`)`	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` $\langle s \rangle$ $\langle action \rangle$ $\langle plan \rangle$`)`

This clause is true for each step $\langle s \rangle$ in plan $\langle plan \rangle$ that uses operator $\langle action \rangle$. 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:

$\langle clause \rangle$ ::= `(define (clause (`$\langle predicate\text{-}name \rangle$ $\langle variable\text{-}name \rangle$`*))`
$\langle function\text{-}body \rangle$`)`

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 $\langle rank \rangle$. 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` $\langle plan \rangle$ $\langle rank \rangle$`)` pattern, the function body of the above definition is called with `plan` and `rank` bound to the $\langle plan \rangle$ and $\langle rank \rangle$ arguments respectively. If `plan-p` accepts $\langle plan \rangle$, 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.

6 Debugging Search Control

In this section we will demonstrate different methods of developing search control rules, focusing mainly on the use of PDB. However, the same basic strategies can be applied from the command line.

6.1 Approaching Search Control

In this section we will cover several different strategies for developing search control using PDB, including an example use of PDB. All of the strategies follow the same basic steps:

1. **Identification**: Recognize a situation where search control might be effective.
2. **Investigation**: Explore the search space in order to determine if and how search control should be used.
3. **Formulation**: Develop a search control rule.
4. **Evaluation**: See if the rule improves planner performance.

This incremental approach is very useful for exploring search control, primarily because of the high potential for interaction between search control rules.

6.2 Debugging Search Control Using PDB

Before reading this section you should be sure to scan the PDB manual which is included as distribution file `doc/pdb.html`.

6.2.1 Identifying Problems

Recognizing potential problems is easy because they happen every time the planner makes any type of decision. For any given problem in a domain, the planner is looking for one solution, but must make choices as to which plans to refine and which flaws to choose. Some choices will lead to solutions faster than others, some will lead to more optimal plans than others, while some choices will not lead to any solutions at all. The hard part is sorting out which decisions lead to poor performance, and which of those decisions can be guided or restricted without sacrificing completeness.

Finding problems is easier if you already have a good idea of what kinds of trouble the planner is having. When you know what you are looking for you can use the VCR Window to scroll through the search tree, searching for a specific refinement or flaw.

If you do not know what the planner may be doing wrong, you might obtain success by exploring the tree in the order in which it was searched. This approach puts you in the role of the planner and may help you understand the problem being solved enough to see where improvements could be made.

Another approach to use when you aren't sure of what is going on is to look for places where the tree branches - wherever there's a branch, there's room for human guidance.

6.2.2 Investigation: Using the Plan Window

The Plan Window allows you to look at plans in detail, as well as giving you the exact flaws chosen for refinement, search control rule firings, and node rankings. Once you have found a point where you think the planner made a bad decision, using a Plan Window will be the easiest way to see if your intuitions are correct.

6.2.3 Investigation: Using the VCR Window

The VCR Window is best for studying groups of nodes. For instance, some phenomena may unfold over several iterations of search rather than at one point in the tree. The VCR Window view gives a higher-level view of this type of situation. Additionally, the VCR Window allows you to estimate the search cost of a particular decision: nodes with many children indicate greater cost than nodes with no children.

6.2.4 Formulating Search Control: Creating a Search Control Rule

The UCPOP search control language supports two types of search control rules: ranking and rejecting. Ranking means mapping plans and flaws to the set of real numbers, always choosing the plan or flaw with the best rank. This type of scheme is very successful when you have good knowledge of how solutions should be structured. Rejecting allows you to stop the planner from ever searching a particular plan. This lends itself to relatively simple search control rules used for situations where the planner is exhibiting obviously inappropriate behavior.

6.2.5 Formulating Search Control: Testing a Search Control Rule

Once you have developed some form of search control, it is important to test it carefully to make sure it works properly - UCPOP doesn't do much in the way of checking rule syntax, so you'll have to take care of "debugging" it yourself. Debugging the rule should come before before worrying about whether or not it improves planner performance. You should start by limiting the search space to a small enough number of nodes that you can fully explore the decisions being made by the planner. This will allow you to make sure that the search controls works the way you intend it to before it is applied to a significant search task.

Search control debugging may also be aided by the use of PDB's plan refinement capabilities (see appendix References). The `Refine` command in the Plan Window's `Plan` menu allows you to direct plan refinement at various levels of detail. Thus you are able to force the planner to make certain decisions and can gauge the resulting effects.

6.2.6 Evaluating the Planner's Performance

Once search control has been debugged, it is time to evaluate the planner's performance. The primary means of comparison will be the number of nodes searched by the planner, but you will also want to factor in the number of nodes actually created (not all of them will have been searched) as well as the CPU time required. If you are solving a set of problems within the same domain, you will want to run extensive tests to make sure that search control is effective across all goals. You may find it helpful to use multiple VCR Windows in order to compare the search trees resulting from different search control strategies.

6.3 Debugging a Search Controller from the Command Line

While the command line interface does not provide the visual capabilities of PDB, you can still follow the basic strategy outline above with the help of speciliazed tracing function.

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)
```

6.4 A Simple Example

6.4.1 Setting up the Example

In this example. we will be investigating the `FIXIT` problem in the `INIT-FLAT-TIRE` domain. This domain is concerned with fixing flat tires on automobiles. The objects in the domain are wheels, nuts, hubs, wrenches, jacks, and containers. Typical problems involve exchanging the wheel on one hub with a different wheel, which in turn requires that the agent fetch the appropriate tools from the trunk, loosen the nuts, jack up the hub, remove the nuts, remove the wheel, etc. `FIXIT` requires not only that the wheel be replaced, but that all tools be back in the trunk at the end of the plan.

We will be using search control to debug the `FIXIT` problem. Specifically, we will try to use the techniques listed above to try to reduce the size of its search tree. The file `"pdb/examples.lisp"` contains some predefined search controllers for use in the example. Please make sure this file is loaded before you begin.

6.4.2 Running `FIXIT`

Create a new session, specifying the `FIXIT` problem, and using search control. The controller you will be using is called `TIRE-CONTROLLER`. It allows the problem to be solved within 2000 nodes. Once planning has completed, bring up a VCR Window on the `FIXIT` session. We are now ready to begin debugging.

6.4.3 Identifying a Problem

For purposes of this tutorial, the potential search control problem has been identified for you.

Scroll to the `START` node and open a Plan Window for that node. Now look at the `START` node and its children from both the VCR Window and from the Plan Window. You'll see that the first flaw UCPOP refined was an open condition: `(:not (open BOOT))`, and that there were two possible refinements.

6.4.4 Investigating with the Plan Window

Using just the Plan Window to explore the tree, you'll see that one child used a `CLOSE` step to support `(:not (open BOOT))` while the other made a Closed World Assumption [11] (`CWA`). You'll also note that the node using `CWA` has lower rank than the other, and was therefore searched first.

6.4.5 Investigating with the VCR Window

You can use the VCR Buttons to locate the `CWA` child of the `START` node, or you can try scrolling the window manually. The VCR Buttons will probably be easier, since PDB's tree formatting algorithm draws the `CWA` node far away from the `START` node. You will see that the `CWA` branch has a substantial subtree beneath it.

6.4.6 Create a Search Control Rule

From the knowledge we gained using the Plan Window, we know that UCPOP used `CWA` to support an open precondition in the goal state. Making the `CWA` is like making a link to the initial state. With such a link in place, any plan which contained a step violating that condition would fail. This is bad for this particular problem since `(open BOOT)` will have to be made true at some point in the plan. We also know from the Plan Window that the `CWA` plan was ranked favorably by our search controller, a fact which is supported by VCR Window navigation. Our use of the VCR Window also shows us that the `CWA` branch was searched quite a bit by UCPOP before being abandoned. Putting all this together, we assume that performance would improve if we could stop UCPOP from searching down this branch.

We want to write a search control rule which states that you should not try to make the Closed World Assumption over the flaw `(:not (open BOOT))` from the goal step. `examples.lisp` contains a rule called `cw-stopper`, which stops the planner from making this decision by increasing the rank of an offending plan by 100.

6.4.7 Test a Search Control Rule

Schedule a new planning session as before, but use `ADVANCED-TIRE-CONTROLLER`[12] as the search controller and set the search limit to 0. When the search limit is set to zero, the resulting session will have only one node in it. The reason you will do this is so that you have a chance to manually interact with the planner and see if the search control rule works before you try to solve the whole problem with it.

Bring up a VCR Window on this new session. The tree should contain only the `START` node. Bring up a Plan Window on this node and select `Refine` from the `Plan` menu. This brings up a dialog which lets you control planning in detail. You can instruct UCPOP to choose the flaw `(:not (open BOOT))` by setting `Choose Flaw` to be true and selecting the appropriate condition from the list. Clicking `Plan` will cause UCPOP to plan down one level

[11] The Closed World Assumption states that any propositions not contained in the initial state are assumed to be false in the initial state.

[12] `ADVANCED-TIRE-CONTROLLER` is just like `TIRE-CONTROLLER` but includes the `cw-stopper` search control rule

using that flaw. You will note that our search control rule works as advertised, increasing the rank of the `CWA` node by 100.

NOTE: You can shortcut this process simply by clicking left on the `(:not (open BOOT))` flaw in the Plan Window itself. This will take you directly to the plan refinement dialog with the selected flaw already chosen.

6.4.8 Evaluate the Planner's Performance

Now plan again. You can either schedule a new session with a 2000 node search limit, or use the test session you've already created. To re-use the old session, go to the `START` node and perform a refinement on it using the 2000 node search limit. UCPOP will plan from this node while PDB records the tree, updating the VCR Window accordingly. When planning is complete, you will see that the new session reduces the search tree considerably.

A 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.

A.1 Switches

`*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.

`*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.

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 [14] for a more detailed explanation of this strategy.

A.2 Global Variables

There are several global variables used to control UCPOP. For instance, the variable `*search-limit*` was already mentioned in section 3.1.3. 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.

There are more (low level) global variables besides these — see the source code.

A.3 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.

- `flaw-fun` This optional keyword argument is a function which must take a single argument (a plan structure) and returns the next flaw that the given plan should solved. The default value is `#'get-flaw` which solves all the threats first, then open conditions. The open conditions are solved according to the order in which steps are added to the plan; more recent open conditions are solved first.

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

A.4 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.

The following bugs are known at this time:

- When compiling ucpop and pdb, Lisp may give warnings about unreferenced functions. These are bogus - Lisp just couldn't figure out the dependencies at compile time.

- (PDB) may conflict with other applications on your desktop for use of a color mapped display. If you run out of colors, you may need to shut down some graphical applications, or `xsetroot` away your background pictures. If `netscape` is the offender, restart netscape with the `-install` option and try again.

- The unification routines used by UCPOP are optimized for speed with the following important assumption — predicates have fixed arity. You may get anomalous results if you violate this assumption. In general, it is an extremely bad idea to use predicates with the same name but different number of arguments since `(predicate A)` may unify with `(predicate ?x ?y)`.

- Do not use the atom "nil" in your domain descriptions.

B 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.

B.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.

B.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.

B.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.

B.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.

B.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.

B.6 Threatened Links

Causal links which are threatened by a third step are represented with structures called `unsafe`s. 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.

B.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.

References

[1] J. Allen, N. Lehrer, M. Boddy, J. Breese, M. Burstein, J. Carciofini, R. Desimone, C. Hammond, J. Lowrance, R. MacGregor, T. Russ, B. Schrag, S. Smith, A. Tate, M. Wellman, and D. Wilkins. Knowledge representation specification language. Technical Report Version 2.0, DARPA / Rome Laboratory Planning and Scheduling Initiative, 1992.

[2] Artificial Intelligence Applications Institute, 80 South Bridge, Edinburgh EH1 1HN, United Kingdom. *O-Plan2 Users Guide*, 1994. Available via anonymous ftp from aiai.ed.ac.uk.

[3] A. Barrett and D. Weld. Task-decomposition via plan parsing. In *Proc. 12th Nat. Conf. on A.I.*, July 1994. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[4] J. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Perez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and tutorial. Cmu-cs-92-150, Carnegie-Mellon University, 1992. Available via anonymous ftp from ftp.cs.cmu.

[5] Foley et. al. *Computer Graphics: Principles and Practice.* Addison Wesley, 1990.

[6] Preece et. al. *Human-Computer Interaction.* Addison Wesley, 1994.

[7] S. Kambhampati and Biplav Srivastava. Universal classical planner: An algorithm for unifying state- space and plan-space planning. Department of Computer Science and Engineering TR-94-002, Arizona State University, January 1995 1995. Submitted to IJCAI.

[8] P. Karp, J. Lowrance, T. Strat, and D. Wilkins. The grasper-cl graph management system. Technical report, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, February 1994. Available via http://www.ai.sri.com/people/wilkins/papers.html.

[9] R. Korf. Linear-space best-first search: Summary of results. In *Proc. 10th Nat. Conf. on A.I.*, pages 533–538, July 1992.

[10] S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. PRODIGY 2.0: The Manual and Tutorial. CMU-CS-89-146, Carnegie-Mellon University, May 1989.

[11] E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[12] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[13] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[14] M. Peot and D. Smith. Threat-removal strategies for partial-order planning. In *Proc. 11th Nat. Conf. on A.I.*, pages 492–499, June 1993.

[15] L. Schubert and A. Gerevini. Accelerating partial order planners by improving plan and goal choices. In *Proceedings of the 7th IEEE Int. Conference on Tools with Artificial Intelligence*, November 1995.

[16] D. Weld. An introduction to least-commitment planning. *AI Magazine*, pages 27–61, Winter 1994. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[17] D. Weld and O. Etzioni. The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. on A.I.*, July 1994. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.