Documentation for SHOP2

Department of Computer Science
University of Maryland
College Park, MD 20742, USA
SIFT, LLC
Minneapolis, MN, USA

Version 1.3. Last update: May 26, 2005.

1 Introduction

This document presents the design and implementation details of SHOP2, the Simple Hierarchical Ordered Planner. SHOP2 adds several extensions to the functionality of SHOP 1.6.1 and MSHOP 1.1.1, and adopts a modified domain syntax for partial compatibility with the recently released Java version of SHOP, which is known as JSHOP 1.0. This document is based, in part, on the JSHOP documentation written by Füsun Yaman, with additional material from Yue Cao's December 2000 draft of the SHOP2 documentation and pseudocode from [Nau *et a*l., 2001]. The entire SHOP research group shares credit for the content of this document, especially Professor Dana S. Nau who wrote the original version of SHOP and its documentation. Additional contributions were made by Robert P. Goldman of SIFT, LLC.

The rest of the document is organized as follows:

- Section one provides an overview of SHOP2 and of the downloadable distribution.
- Section two describes the environment for execution of SHOP2.
- Section three explains the typographic conventions used in this document.
- Section four presents the formalism used for inputs and outputs of SHOP2.
- Section five provides instructions for running SHOP2.
- Section six presents internal technical details regarding SHOP2.
- Section seven presents the Java interface for SHOP2.
- Section eight presents the SHOP2 Graphical User Interface.
- Sections nine and ten summarize the differences between SHOP2 and SHOP 1.x and JSHOP 1.0, respectively.
- Section eleven presents some general remarks on the SHOP2 system.
- Sections twelve and thirteen list acknowledgements and references, respectively.

1.1 Overview

SHOP (Simple Hierarchical Ordered Planner) is a domain-independent planning system based on **ordered task decomposition**, a modified version of HTN planning that involves planning for tasks in the same order that they will later be executed. SHOP has the following characteristics:

- SHOP knows the current state-of-the-world at each step of the planning process.
- It has large expressive power. For example, in the preconditions of operators and methods it can do mixed symbolic/numeric computations and execute calls to external programs.
- SHOP can be used to create very efficient domain-specific planning algorithms. The SHOP software distribution includes several examples of such domain algorithms.
- An earlier version of the SHOP algorithm implemented in Java is used as part of HICAP, a plan-authoring system for complex military operations.
- SHOP2 incorporates many features from PDDL, e.g., support for quantifiers and conditional effects in methods and operators.
- SHOP2 allows the combination of partially ordered and fully ordered tasks through the use of the `:unordered` and `:ordered` keywords.
- SHOP2 allows branch-and-bound optimization of plan costs. For small problems, this capability can be used to find the absolute minimum cost plans. For larger problems, this capability can be used with time limits to get the lowest cost plan that is found within the given time limit.

2 Execution Environment

SHOP2 is written in Common Lisp. To be able to run SHOP2, you will need to have Common Lisp installed on your computer. We have run SHOP2 successfully under the following implementations of Common Lisp, and we would be interested in hearing your reports about other implementations:

- Allegro Common Lisp (on Sun Sparcstations running Solaris);
- Allegro Common Lisp v.5.01 (on Intel PC's running Windows 2000);
- Xanalys Lispworks v. 4.1 (on Sun Sparcstations running Solaris)
- Macintosh Common Lisp (on Macintosh PowerPC's running Mac OS).

SHOP2 has been tested most heavily on the first of these (Allegro / Solaris). The software distribution is in ZIP format. Freeware ZIP decoders are available for many different platforms, from sites such as Info-Zip, CAM Development, and Aladdin Systems.

Important Performance Note: Make sure to compile the main Lisp file provided in the SHOP2 distribution before running SHOP2. If you don't, the program will run *very* slowly. Some Common Lisp implementations (e.g., Macintosh Common Lisp) will compile the files for you automatically when you load them. Other implementations (e.g., Allegro Common Lisp) will require you to compile the files manually.

3 Notations Used in This Document

In order to differentiate some words or expressions in the text, we used the following conventions:

- Boldface is used to indicate that a term is being defined. For example: "An **axiom list** is a list of axioms intended to represent what we can infer from a state."
- Italic characters refer to special words or symbols. For example: " Let *a* be a *logical atom*."
- Typewriter characters are used to write computer code. For example: "`(call <= 7 (call  + 5 3))`"
- Square brackets indicate that a parameter or keyword is optional. For example, in the following form, the $name_i$'s are optional parameters and thus the form is still valid if any of the $name_i$'s are missing:
  "(:- a [$name_1$] C_1 [$name_2$] C_2 [$name_3$] C_3 ... [$name_n$] C_n)"

4 The SHOP2 Formalism

The inputs to SHOP2 are a *planning domain* and either a single *planning problem* or a *planning problem set*. Planning domains are composed of *operators*, *methods*, and *axioms*. Planning problems are composed of *logical atoms* (an initial state) and *tasks lists* (high-level actions to perform). Planning problem sets are composed of planning problems.

The components of a planning domain (operators, methods, and axioms) all involve *logical expressions*. These logical expressions combine *logical atoms* through a variety of forms (e.g., conjunction, disjunction). Logical atoms involve a *predicate symbol* plus a list of *terms*. Task lists in planning problems are composed of *task atoms*. The components of domains and problems are all ultimately defined by various *symbols*.

This section describes each of the aforementioned structures. It is organized in a bottom-up manner because the specification of higher-level structures is dependent on the specification of lower-level structures. For example, methods are defined after logical expressions because methods contain logical expressions.

4.1 Symbols

In the structures defined below, there are five kinds of symbols: **variable symbols**, **constant symbols**, **function symbols**, **primitive task symbols**, and **compound task symbols**. To distinguish among these symbols, SHOP and SHOP2 both use the following conventions:

- a **variable symbol** can be any Lisp symbol whose name begins with a question mark (such as `?x` or `?hello-there`)
- a **primitive task symbol** can be any Lisp symbol whose name begins with an exclamation point (such as `!unstack` or `!putdown`)
- a **constant symbol**, a **function symbol**, a **predicate symbol**, or a **compound task symbol** can be any Lisp symbol whose name does not begin with a question mark or exclamation point

Any of the structures defined in the remaining sections are said to be **ground** if they contain no variable symbols.

4.2 General Lisp Expressions

A number of SHOP2 domain structures described in this section use **general Lisp expressions**. These are arbitrary pieces of Lisp code which can include functions, macros, special macro symbols (e.g., backquote), etc. When SHOP2 needs to get the value of a general Lisp expression, it first substitutes values for any variable symbols in the expression that are bound. Then it sends the entire expression into the Lisp evaluator to get a final value.

Note: Counter-intuitive bugs may arise when symbols are passed to lisp for evaluation (either as constants or as the values of variables). Remember that the lisp evaluator will assume that these are variables! If you wish them to be treated as symbols, you will need to quote them. This leads to a slightly undesirable oddity --- variables that will be bound to, for example, numbers, can appear normally. Variables that will be bound to symbols will have to be quoted. See the discussion of Eval terms, below (Section 4.3.2).

4.3 Terms

A **term** is any one of the following:

- a variable symbol
- a constant symbol
- a number
- a **list-term**
- an **eval-term**
- a **call-term**

4.3.1 List Terms

A **list-term** is a term having the form

`([list]` t_1 t_2 ... t_n *[. l]*`)`

where *list* is an optional reserved word and each t_i is a term. This specifies that t_1 t_2 ... t_n are the items of a list. If the final, optional element is included, the item *l* should evaluate to a list; all items in *l* are included in the list after t_1 through t_n.

4.3.2 Eval Terms

An **eval-term** is an expression of the form

`(eval` *general-lisp-expression*`)`

The value associated with an eval-term is determined as follows. First, any variable symbols which appear in *lisp-expression* and are bound are replaced by the values that they are bound to. Then, the entire expression is evaluated in Lisp. For example, if the variable symbol `?foo` is bound to the value `3` then the term:

```
(eval (mapcar #'(lambda (x) (+ x ?foo)) `(1 2 ,(* ?foo ?foo))))
```

will have as its value a list containing the numbers 4, 5, and 12. Note that variable substitutions in eval terms are handled before any evaluation of the expression, as in Lisp macros. One implication of this fact is that variables with symbolic values must be explicitly quoted if they are to be treated as Lisp symbols. For example, if the variable `?foo` is bound to the symbol `BAR`, the following eval term has the value `(BAR BAZ)`:

```
(eval (list '?foo 'BAZ))
```

if this were written

```
(eval (list ?foo 'BAZ))
```

it would cause a Lisp error when lisp attempts to find the value of `BAR`, which it would believe to be variable.

4.3.3 Call-terms

A **call-term** is an expression of the form

(call f t_1 t_2 ... t_n)

where `f` is a function symbol and each t_i is a term or a call-term. A call-term has a special meaning to SHOP2, because it tells SHOP2 that `f` is an attached procedure, i.e., that whenever SHOP2 needs to evaluate a precondition or task list that contains a call-term, SHOP2 should replace the call term with the result of applying the function `f` on the arguments t_1, t_2, …, t_n. (We later will define what preconditions and task lists are).

For example, the following call-term would have the value 6:

```
(call  +  (call + 1 2)  3)
```

Note that a call-term is not as expressive as an eval-term. In particular, it does not support the evaluation of Lisp macros (including macro characters such as backquote). Both `call` and `eval` are supported in SHOP2 because the former is compatible with JSHOP 1.0 and the latter is compatible with SHOP 1.x. SHOP2 users who are not interested in either form of compatibility may use either form.

4.4 Logical Atoms

A **logical atom** has the form:

(p t_1 t_2 ... t_n)

where p is a predicate symbol and each t_i is a term other than an eval- or call-term.

4.5 Logical Expressions

A **logical expression** is a logical atom or any of the following complex expressions: **conjuncts**, **disjuncts**, **negations**, **implications**, **universal quantifications**, **assignments**, **eval expressions**, **call expressions**.

4.5.1 Conjuncts

A **conjunct** has the form

`([and]`l_1 l_2 `...` l_n`)`

where each l_i is a logical expression. Note that if there are 0 conjuncts (e.g., the expression is `()`) then the form always evaluates to true.

4.5.2 Disjuncts

A **disjunct** is an expression of the form

`(or` l_1 l_2 `...` l_n`)`

where l_1, l_2,..., l_n are logical expressions.

4.5.3 Negations

A **negation** is an expression of the form

`(not` l`)`

where l is a logical expression.

4.5.4 Implications

An **implication** is an expression of the form

`(imply` Y Z`)`

where Y and Z are logical expressions. The intent of an implication is to evaluate its logical counterpart; that is, $(\neg Y \vee Z)$. Note that here, Y should be ground, or the semantic of the implication will be ambiguous.

4.5.5 Universal Quantifications

A **universal quantification** expression is an expression of the form

`(forall` V E_1 E_2`)`

where E_1 and E_2 are logical expressions, and V is the list of variables in E_1. To satisfy a **universal quantification** expression, the following must hold: for each possible substitution u for variables in V, if E_1^u is satisfied then E_2^u must also be satisfied in the current state of the world. Note that this use of the keyword "forall" is distinct from its use in add and delete lists in operators (see Section 4.10); the latter is used to express a set of effects rather than a logical expression and consequently has a different syntax.

4.5.6 Assignments

An **assignment** expression has the form

```
(assign v e)
```

where *v* is a variable symbol and *e* is general Lisp expression. The intent of an assignment expression is to bind the value of *e* to the variable symbol *v*. Variable substitutions in assignment expressions are done using literal substitutions, as with eval terms (see Section 4.3.2). For example, if `?foo` is bound to the symbol `IF` and `?bar` is bound to the number 0 then the following expression will bind the variable `?baz` to the list `(IF FISH)`:

```
(assign ?baz (?foo (< ?bar 3) (list '?foo 'fish) (/ 8 ?bar)))
```

Similarly, if `?foo` is bound to `LIST` and `?bar` is bound to 4 then the expression above will bind `?baz` to the list `(NIL (LIST FISH) 2)`.

4.5.7 Eval expressions

An **eval-expression** has the same form as an eval-term, q.v. Unlike an eval-term, however, an eval-expression is interpreted simply as either true or false rather than having some value which would be used as an argument to a predicate. Thus eval-expression typically invoke boolean Lisp functions such as `evenp` or >=.

4.5.8 Call-expressions

A **call-expression** has the same form as a call-term, q.v. As with call-expressions, eval-expressions are interpreted as true or false.

4.6 Logical Precondition

A **logical precondition** is a either logical expression or one of the following special precondition forms: **first satisfier precondition**, **sorted precondition**.

4.6.1 First Satisfiers Precondition

A **first satisfier precondition** has the form

$$(\texttt{:first}\ l_1\ l_2\ ...\ l_n)$$

where each l_i is a logical expression. Such a precondition causes SHOP2 to consider only the first set of bindings that satisfies all of the given expressions. Alternative bindings will not be considered even if the first bindings found do not lead to a valid plan.

4.6.2 Sorted Precondition

A **sorted precondition** has the form

```
(:sort-by ?v [e] l)
```

where *?v* is a variable symbol, *e* is a general Lisp expression (which should evaluate to a comparison function), and *l* is a logical expression. Such a precondition causes SHOP2 to consider bindings for the precondition in a specific order. Specifically, bindings are sorted such that if the specified comparison function holds between values *x* and *y* then bindings that bind *?v* to *x* may not occur after bindings that bind *?v* to *y*. For example consider the precondition:

```
(:sort-by ?d #'> (and (at ?here) (distance ?here ?there ?d)))
```

This precondition will cause SHOP2 to consider bindings in decreasing (high to low) order of the value of *?d*. If the comparison function (*e*) is omitted, it defaults to `#'<`, indicating increasing (low to high) order.

4.7 Axioms

An **axiom** is an expression of the form

$$(\texttt{:- a [name}_1\texttt{]}\ E_1\ \texttt{[name}_2\texttt{]}\ E_2\ \texttt{[name}_3\texttt{]}\ E_3\ \ldots\ \texttt{[name}_n\texttt{]}\ E_n)$$

where the axiom's **head** is the symbol *a*, and its **tail** is the list ([$name_1$] E_1 [$name_2$] E_2 [$name_3$] E_3 ... [$name_n$] E_n) and each E_i is a logical expression and each $name_i$ is a symbol called the *name* of E_i. The names of the expressions are optional. When a domain definition is loaded into SHOP2, a unique name will be generated for each conjunct if no name was given. These names have no semantic meaning to SHOP2, but are provided to help the user debug domain descriptions by looking at traces of SHOP2's behavior.

The intended meaning of an axiom is that *a* is true if E_1 is true, or if E_1 is false but E_2 is true, or if all of E_1, E_2, ... , E_{n-1} are false but E_n is true. For example, the following axiom says that a location is in walking distance if the weather is good and the location is within two miles of home, or if the weather is not good and the location is within one mile of home:

```
(:- (walking-distance ?x)
     good   ((weather-is good) (distance home ?x ?d) (call <= ?d 2))
     bad    ((distance home ?x ?d) (call <= ?d 1)))
```

4.8 Task Atoms

A **task atom** is an expression of any of the forms

$$(s\ t_1\ t_2\ \ldots\ t_n)$$
$$(\texttt{:task}\ s\ t_1\ t_2\ \ldots\ t_n)$$
$$(\texttt{:task :immediate}\ s\ t_1\ t_2\ \ldots\ t_n)$$

where *s* is a task symbol and the arguments t_1, t_2, ..., t_n are terms. The task atom is **primitive** if *s* is a primitive task symbol, and it is **compound** if *s* is a compound task symbol. The first and second forms are called an **ordinary task atom**; the third form is called an **immediate task atom**. The purpose of the `:immediate` keyword is to give a higher priority to the task, as described in the following subsection.

4.9 Task Lists

A **task list** is any of the following:

- an expression of the form (`:unordered` $tasklist_1$ $tasklist_2$ ... $tasklist_n$), where $tasklist_1$ $tasklist_2$ ... $tasklist_n$ are task lists;
- an expression of the form ([`:ordered`] $tasklist_1$ $tasklist_2$ ... $tasklist_n$), where $tasklist_1$ $tasklist_2$ ... $tasklist_n$ are task lists.

The `:ordered` keyword, which is optional, specifies that SHOP2 must perform the task lists in the order that they are given. The `:unordered` keyword specifies that there is no particular ordering specified between $tasklist_1$, $tasklist_2$ ... $tasklist_n$. With the use of the `:unordered` keyword, SHOP2 may interleave tasks between different task lists. Suppose we have two task lists as the following:

T = (`:ordered` $t_1\, t_2$ ... t_m);
U = (`:ordered` $u_1\, u_2$ ... u_n);

and that we have the main task list

M = (`:unordered` $T\ U$).

If none of the tasks have the `:immediate` keyword, then the tasks in T should be performed in the order given, and the tasks in U should also be performed in the order given—but it is permissible for SHOP2 to interleave the tasks of T and the tasks of U. However, if some of the tasks are immediate, then each time SHOP 2 chooses the next task to accomplish, it needs to give a higher priority to the immediate tasks. For example, if t_1 is immediate and u_1 is not immediate, then SHOP2 should perform t_1 before both t_2 and u_1.

Note: A task with the `:immediate` keyword specifies that this task must be performed immediately when it has no predecessors. Therefore, we can allow only one task with the `:immediate` keyword in the list of tasks that have no predecessors. Otherwise, SHOP2's behavior on those tasks is undefined. In other words, it is not allowed to have two tasks in an `:unordered` list and both have the `:immediate` keyword. For instance, on the example above, t_1 and u_1 cannot both have the `:immediate` keyword.

4.10 Operators

An **operator** has the following form:

(`:operator` *h P D A [c]*)

where

- *h* (the operator's **head**) is a primitive task atom (i.e., a task atom with a task symbol that begins with an exclamation point)
- *P* (the operator's **precondition**) is a logical expression.
- *D* (the operator's **delete list**) is a list for which each of the element may be any of following:
  - a logical atom
  - a protection condition (see below)
  - an expression of the form `(forall V E L)`, where *V* is a list of variables in *E*, *E* is a logical expression, and *L* is a list of logical atoms
- *A* (the operator's **add list**) is a list of logical atoms that has the same form as D.
- *c* (the operator's **cost**) is a general Lisp expression. If *c* is omitted, the cost is 1.

For backwards compatibility with SHOP 1.x, SHOP2 will also accept operators where the precondition *P* is missing. In this case the domain definition pre-processing code puts a null precondition into the operator, which is always satisfied. SHOP2's ability to recognize operators without preconditions will be deprecated and is likely to disappear in the future.

In the above definition, a **protection condition** is an expression of the form

```
(:protection a)
```

where *a* is a logical atom. The purpose of a protection condition in the add list is to tell SHOP2 that it should not execute any operator that deletes *a*. The purpose of a protection condition in the delete list is to cancel a previously added protection condition. For example, if we drive a delivery truck to a certain location in order to pick up a package, then we might not want to allow the truck to be moved away from that location until after we have picked up the package. To represent this, we might use the following operators:

```
(:operator (!drive-to ?truck ?old-loc ?location)
      ()
      ((at ?truck ?old-loc))
      ((at ?truck ?location)
            (:protection (at ?truck ?location))))

(:operator (!pick-up ?truck ?package ?location)
      ()
      ((at ?package ?location)
            (:protection (at ?truck ?location)))
      ((in ?package ?truck)))
```

As noted above, the head of the operator is a primitive task atom, so it must begin with a primitive task symbol, i.e., a symbol that begins with an exclamation point. Note that operator names which begin with *two* exclamation points have a special meaning in SHOP2; operators of this sort are known as **internal operators**. Internal operators are ones which are used for purposes internal to the planning process and are not intended to correspond to actions performed in the plan (e.g., to do some computation which will later be useful in deciding what actions to perform). Other than requiring two exclamation points at the start of the name, the syntax for internal operators is identical to the syntax for other operators. SHOP2 handles internal operators exactly the same way as ordinary operators during planning. SHOP2 includes these operators in any plans that it returns at the end of execution. It may, however, omit them from the printout of the final plan (depending on the value of the `:verbose` argument described in Section 5.1). The primary reason that the internal operator syntax exists in SHOP2 is so that automated systems which use SHOP2 plans as an input can easily distinguish between those operators which involve action and those which were merely internal to the planning process.

When designing an operator, it is important to ensure that each variable symbol in the add list, delete list, and cost always be bound to a single value when the operator is invoked. Variable symbols can be bound in the head of the operator (by the method that invokes the associated primitive task) or in the precondition of the operator. An operator should be written such that for any variable appearing after the precondition, no two unifiers of the precondition have different bindings for that variable. SHOP2 does not check this requirement; if conflicting unifiers are available when applying an operator, it will apply one arbitrarily. This can lead to unpredictable behavior and plans with ambiguous semantics. In general, we recommend that operator preconditions be designed such that only one unifier is possible. However, SHOP2 will be able to correctly process operators that have multiple unifiers for preconditions as long as no two unifiers can provide different values for a variable that appears in the add list, delete list, or cost.

4.11 Methods

A **method** is a list of the form

$$\texttt{(:method } h\ \ [n_1]\ C_1\ T_1\ [n_2]\ C_2\ T_2\ \ldots\ [n_k]\ C_k\ T_k\texttt{)}$$

where

- h (which is called the method's **head**) is a task atom in which no *call- or eval-terms* can appear;
- Each C_i (which is called a **precondition** for the method) is a logical precondition.
- Each T_i (which is called a **tail** of the method) is a task list. The task atoms in the list can contain call-terms.
- Each n_i is the *name* for the succeeding C_i T_i pair. These name are optional and if omitted a unique name will be assigned for each pair. These names have no semantic meaning to SHOP2, but are provided in order to help the user debug domain descriptions by looking at traces of SHOP2's behavior.

A method indicates that the task specified in the method's head can be performed by performing all of the tasks in one of the methods tails when one that tail's precondition is satisfied. Note that the preconditions are considered in the given order, and a later precondition is considered *only* if all of the earlier preconditions are not satisfied. If there are multiple methods for a given task available at some point in time, all of these methods can be considered. Consequently, the following code:

```
(:method (eat ?food)
      (have-fork ?fork)
      ((!eat-with-fork ?food ?fork))
      (have-spoon ?spoon)
      ((!eat-with-spoon ?food ?spoon))
```

is semantically equivalent to the following code with multiple methods and explicitly exclusive preconditions:

```
(:method (eat ?food)
      (have-fork ?fork)
      ((!eat-with-fork ?food ?fork)))
(:method (eat ?food)
      (and (not (have-fork ?fork)) (have-spoon ?spoon))
      ((!eat-with-spoon ?food ?spoon))
```

In both of the above examples, the !eat-with-spoon operator may be performed only if the `(have-spoon ?spoon)` is satisfied *and* `(have-fork ?fork)` is not satisfied.

4.12 Planning Domain

A **planning domain** has the form

`(defdomain` *domain-name* `(`i_1 i_2 ... i_n`))`

where *domain-name* is a symbol and each item i_i is one of the following: an operator, a method, or an axiom. Note that domain names are not used in SHOP2; they are left in the syntax for backward compatibility.

4.13 Planning Problem

A **planning problem** has the form

`(defproblem` *problem-name domain-name* `(`a_1 a_2 ... a_n) T)

where *problem-name* is a symbol, *domain-name* is a symbol, each a_i is a ground logical atom, and T is a task list. This form defines a problem which may be solved by addressing the tasks in T with the initial state defined by the atoms a_1 through a_n.

4.14 Planning Problem Set

A **planning problem set** has the form

`(def-problem-set` *set-name* `(`p_1 p_2 ... p_n`))`

where *set-name* is a symbol and each p_i is the name of a planning problem.

4.15 Plans

The previous subsections describe the inputs to SHOP2. This subsection describes the result that SHOP2 produces. A **plan** is a list of the form

$(h_1\ \ c_1\ \ h_2\ \ c_2\ \ \ldots\ \ h_n\ \ c_n)$

where each h_i and c_i, respectively, are the head and the cost of a ground operator instance o_i. If $p = (h_1\ \ c_1\ \ h_2\ \ c_2\ \ \ldots\ \ h_n\ \ c_n)$ is a plan and S is a state, then $p(S)$ is the state produced by starting with S and executing $o_1, o_2, \ldots, o_n$ in the order given. The **cost** of the plan p is $c_1 + c_2 + \ldots + c_n$ (thus, the cost of the empty plan is 0).

5 Running SHOP2

There are two ways to execute the SHOP2 planning process: `find-plans`, which finds plans for a single planning problem, and `do-problems`, which finds plans for a planning problem set. The first subsection below describes the use of these functions. The next subsection describes the functions `shop-trace` and `shop-untrace`, which are the primary mechanisms for debugging SHOP2 domain descriptions and problem specifications. The third subsection describes some additional features that may also be useful for debugging domain descriptions and problems for SHOP2. Finally, the fourth subsection describes some hook routines that can be used to customize the behavior of SHOP2.

5.1 Executing the Planner

The `find-plans` function has one mandatory argument, the name of a planning problem, and a set of optional keyword arguments. It returns up to four values. `Find-plans` will always return two values: (1) a list of plans and (2) the total amount of CPU time used during planning (in seconds). If the :plan-tree argument (see below) is non-NIL, then two additional values will be returned: (3) a list of plan tree data structures and (4) a list of final state data structures. From the plan state data structures, the user can extract full state trajectories for the plans.

The `do-problems` function has one mandatory argument, which can either be the name of a planning problem set or a list of names of planning problems. It executes `find-plans` on each of the given planning problems and returns `nil`. Both of these functions use the same keyword arguments.

The keyword arguments to `find-plans` and `do-problems` are as follows:

- *which* says what kind of search to do. Here are its possible values and what they mean. The default value of *which* is the value of the global variable `*which*` (whose default value is `:first`).

Value	**Kind of search**
`:first`	Depth first search, stopping at the first plan found
`:all`	Depth-first search, but don't stop until all plans in plans(S, T, M) have been found
`:shallowest`	Depth-first search for the shallowest plan (or the first such plan if there is more than one of them). In many domains, this is also the least-cost plan

`:all-shallowest`	Depth-first search for all shallowest plans in the search space
`:id-first`	Iterative-deepening search, stopping after the first plan found
`:id-all`	Iterative-deepening search for all shallowest plans

The `:id-all` and `:id-first` options are equivalent to taking a modified version of `find-plans` that backtracks each time it reaches depth *d*, and calling it repeatedly with *d* = 1, 2, ..., until a plan is found.

- *verbose* says what information to print out, as shown in the following table. The default value for *verbose* is 1.

Value	**What to print**
`0` or `nil`	Nothing
`1` or `:stats`	Some statistics about the search
`2` or `:plans`	The statistics plus a succinct version of each plan found (internal operators and operator costs are omitted).
`3` or `:long-plans`	The statistics plus the complete version of each plan found

- If *gc* is non-`nil`, then `find-plans` calls the garbage collector just before starting its search, thus making it somewhat easier to get repeatable experimental results. Note that this feature of SHOP2 is only supported under Macintosh Common Lisp and Allegro Common Lisp; in all other Lisp implementations the value for this keyword is ignored. The default value of *gc* is `t`.
- If *pp* is non-`nil`, then all printing done by SHOP2 is performed using the Common Lisp pretty-printing mechanism. This typically leads to more easily read output. The default value of *pp* is `t`.
- The *state* argument controls how states are represented internally. SHOP2 can have different performance characteristics depending on the value provided to this augment. If you are encountering out-of-memory errors in SHOP2 or you want to get the maximum speed possible from SHOP2 for a particular set of problems, you may wish to experiment with different values for this argument. The default value is `:mixed`, which represents states using a combination of lists and hash tables; this value has been shown to provide a reasonably good combination of speed and memory usage on a variety of test problems. The other values are `:list`, `:hash`, and `:bit`.
- The *optimize-cost* argument is used to perform planning with branch-and-bound optimization of the total plan cost. The default value for this argument is `nil`. If the value of this argument is `nil`, the optimization feature is disabled. If the value of the argument is t, SHOP2 will search for plans with the minimum total cost. If

the value of the argument is a number, SHOP2 will use the branch-and-bound technique to search for plans with cost less than or equal to the value of the argument. The optimization feature is written under the assumption that the costs of operators are always non-negative. If this assumption is invalid, SHOP2 will produce unreliable results (specifically it will prune out some valid plans). The interaction of `:optimize-cost` with the various options for :which can be subtle. Below are notes on each possible combination:

- `(:which :first :optimize-cost t)`
  Under these arguments, SHOP2 returns the first plan found for which no other valid plan has a lower total cost. Note that this option may take much more time to run than using `(:which :first :optimize-cost nil)` since even after it finds the plan, it must keep searching to see if it can find a cheaper plan. However, this option may be significantly faster than `(:which :all :optimize-cost nil)` since the branch-and-bound mechanism will prune out non-optimal plans without having to consider them all the way to the end. In some cases, this will mean that `(:which :first :optimize-cost t)` terminates and `(:which :all :optimize-cost nil)` does not.
- `(:which :first :optimize-cost` *`number`*`)`
  Under these arguments, SHOP2 returns the first plan found whose total cost is less than or equal to the number given. If there is no plan whose total cost is less than or equal to that number, SHOP2 will return no plans. Note that if the number given is large enough, these arguments can produce results much more quickly than with `(:which :first :optimize-cost t)`; specifically, as soon as SHOP2 finds a plan for which the cost is met, it can terminate and does not have to keep searching for cheaper plans.
- `(:which :all :optimize-cost t)`
  Under these arguments, SHOP2 returns all plans for which no other valid plan has a lower total cost. Obviously, all plans returned under these options will have equal total cost.
- `(:which :all :optimize-cost` *`number`*`)`
  Under these arguments, SHOP2 returns all plans with total cost less than or equal to the given number.
- `(:which :shallowest :optimize-cost t)`
  Under these arguments, SHOP2 returns a plan that has the shallowest depth of all valid plans and for which there is no other shallowest depth valid plan which has a lower total cost. In other words, these arguments produce the cheapest of all shallowest plans (which, incidentally, is not necessarily the same thing as the shallowest of all cheapest plans).
- `(:which :shallowest :optimize-cost` *`number`*`)`
  Under these arguments, SHOP2 returns a plan which has the shallowest depth of all valid plans and whose total cost is less than or equal to the given number. Note that if there is no plan whose cost is less than or equal to the number and whose depth is shallowest among all valid plans, then no plan will be returned (even if there are deeper plans which do have cost less than or equal to the number).

- o `(:which :all-shallowest :optimize-cost t)`
  Under these arguments, SHOP2 returns all plans which have the shallowest depth of all valid plans and for which there is no other shallowest depth valid plan which has a lower total cost.
- o `(:which :all-shallowest :optimize-cost` *`number`*`)`
  Under these arguments, SHOP2 returns all plans which have the shallowest depth and whose total cost is less than or equal to the given number.
- o `(:which :id-first)` or `(:which :id-all)`
  The *id-first* and *id-all* arguments produce the same results as the *shallowest* and *all-shallowest* arguments, respectively for each different combination with `:optimize-cost`. Note, however, that there are domains for which SHOP2 will terminate using *id-first* and *id-all* and will not terminate using other values for `:which`.

- The *time-limit* argument may either `nil` or a number. It's default is `nil` and if it is `nil`, no time limit is imposed on the planning process. If the *time-limit* argument is a number, SHOP2 will check the elapsed CPU time at the start of each step of the planning process, and if the number of seconds elapsed is greater than the argument value, SHOP2 will immediately terminate. The main use for this feature is in combination with `(:optimize-cost t)` argument, in order to return the optimal value found within the given time limit. For example, consider the call `(find-plans 'foo :verbose 1 :optimize-cost t :time-limit 120)`. This call addresses a problem named *foo*, and runs until it either finds the minimum cost plan or until 2 minutes have elapsed. It then returns the lowest cost plan that it found during that time. This functionality is inspired, in part, by Anytime Algorithms [Dean and Boddy, 1998].
- If *explanation* is non-nil, SHOP2 adds extra information at the end of each operator explaining how the preconditions for that operator were satisfied. Currently supports only logical atoms, `and`, and `or`; it doesn't work with `forall`, `not`, `eval`, etc. If this feature is used with the `external-access-hook` feature (see Section 5.4), any attribution information provided by the `external-access-hook` routine is included in the relevant explanation. The default value of *explanation* is `nil`.
- The *plan-tree* argument defaults to nil; if true, the planner will return two additional values: (1) a list of complete task decomposition trees for the plans and (2) a list of plan state data structures corresponding to the final states of each plan. Plan trees are encoded in a nested list format in which the decomposition of an upper level task into lower level tasks is represented by the upper level task atom, followed by trees for each lower level task. The leaves of the tree, involving operators, are each lists of three elements: the cost of the operator, the task atom for the operator, and the numerical position of the operator in the plan (staring at 1). For example, a task `(travel houston springfield)` that was directly decomposed into operators, `(!fly houston boston)` with cost 200 and `(!drive boston springfield)` with cost 50, would have the following plan tree:

```
((travel houston sprinfield)
 (200 (!fly houston boston) 1)
```

```
(50 (!boston springfield) 2))
```

5.2 Tracing

There are two functions used for controlling the tracing mechanism in SHOP2: `shop-trace` and `shop-untrace`. These are similar to Lisp's `trace` and `untrace` functions. Once they have been invoked, subsequent calls to `find-plans` or `do-problems` will print out information about elements of the domain for which tracing is enabled whenever those elements are encountered. More specifically:

- `(shop-trace` *`item`*`)` will turn on tracing for item, which may be any of the following:
  - a method, axiom, operator, task, or goal;
  - one of the keywords `:methods`, `:axioms`, `:operators`, `:tasks`, `:goals`, or `:protections` in which case SHOP2 will trace *all* items of that type (`:goals` refers to predicates that are goals for the theorem-prover, and `:protections` refers to predicates used as arguments of `:protection` in operators);
  - the keyword `:states`, in which case SHOP2 will include the current state whenever it prints out a tracing message
  - the keyword `:plans` in which case SHOP2 will print diagnostic information whenever it has found a plan (and may be considering whether or not to keep the plan, depending on the `:which` and `:optimize` arguments of `seek-plans`).
- `(shop-trace` *`item1 item2`* `...)` will do the same for a list of items
- `(shop-trace)` will print a list of what's currently being traced
- `(shop-untrace` *`item`*`)` will turn off tracing for an item
- `(shop-untrace` *`item1 item2`* `...)` will turn off tracing for a list of items
- `(shop-untrace)` will turn off tracing for all items

5.3 Other Debugging Features

There are three variables, namely *current-state*, *current-plan*, and *current-tasks*, in SHOP2. These variables can be used to monitor the current status of the state, current plan and the list of current tasks to be accomplished respectively. Since these are the internal variables of the SHOP2 planning system, the following functions are defined to access the current contents of those variables: `print-current-state`, `print-current-plan`, and `print-current-tasks`, respectively. Note that these are Lisp functions that must be called by using the Lisp evaluator. The best way to use these functions is to define dedicated methods in the domain that invoke the functions using eval or call expressions in their predicates. Those methods can then be used in the problem definition where debugging output is needed. For example, the following methods can be included in any domain description for this purpose:

```
(:method (print-current-state)
      ((eval (print-current-state)))
      ())
```

```
(:method (print-current-tasks)
      ((eval (print-current-tasks)))
      ())
(:method (print-current-plan)
      ((eval (print-current-plan)))
      ())
```

And these special purpose methods can be used in the task decompositions of other methods for debugging purposes. For example,

```
(:method (do-both ?x ?y) nil
      (:ordered (:task !do ?y)
                (:task print-current-state)
                (:task !do ?x))))
```

There is now a new variable, `*break-on-backtrack*`, that will cause the Lisp environment to throw into a break loop when SHOP2 backtracks.

5.4 Hook Routines

SHOP2 recognizes several different hook routines. These are Lisp routines that may be defined by the user; if they are defined, they are invoked under specific circumstances. Hook routines are typically used when embedding SHOP2 in an application; they allow such an application to obtain additional information from SHOP2 or to affect it's behavior. There are three hooks that are recognized by SHOP2:

- `(plan-found-hook state which plan cost depth)`
  If this routine is defined, SHOP2 invokes it whenever it finds a plan. It can be useful for displaying and/or recording details about the plan. The arguments are the current state, the value for the `:which` argument that was provided to the planner, the plan, the cost of the plan, and the search depth at which the plan was found.
- `(trace-query-hook type item additional-information state-atoms)`
  If this routine is defined, SHOP2 invokes it whenever it invokes the tracing mechanism (See Section 5.2). The arguments include the type of item being traced (e.g., `:task`, `:method`), the item, the list of Lisp values that are printed by the tracing mechanism, and a list of logical atoms in the current state.
- `(external-access-hook query)`
  This hook routine is intended to allow SHOP2 to use an external source (such as a database) to determine the applicability of methods and operators. To use this hook routine, a domain must include one or more logical expressions that have the keyword `:external` as the first symbol. Such expressions must only involve a single logical atom, or a single conjunction of logical atoms. When SHOP2 attempts to find a binding that satisfies such an expression, it will first invoke `external-access-hook` to satisfy the expression; if that routine is undefined or returns `nil`, SHOP2 will then try to satisfy the expression using its internal knowledge state. The argument to `external-access-hook` is a list of the form `'(and (<pred> <val> <val>)...)`. It returns a list of responses, each of which

is a list of two elements: an attribution and a list of bindings for the unbound variables in the query. The attribution is stored for use with the *explanation* option for the planning system (See Section 5.1). For example, consider a method that has the following precondition:

```
(or (and (clear ?b1) (clear ?b2) (clear ?b3)
     (:external and (on ?b1 ?b2) (on ?b2 ?b3)))
```

When this precondition is encountered and `external-access-hook` is defined, SHOP2 invokes that routine with the argument `'(and (on ?b1 ?b2) (on ?b2 ?b3))`. The routine might (for example) return the list:

```
'((database-123 ((?b1 block10) (?b2 block 11) (?b3 block 12)))
  (database-223 ((?b1 block20) (?b2 block 21) (?b3 block 22))))
```

6 Internal Technical Information

This section presents information about the internal workings of the SHOP2 planning process. **Important Note**: This section is primarily of interest to planning researchers and planning system developers. Most SHOP2 users (especially beginning users) are advised to skip this section.

The first subsection presents some key internal knowledge structures which must be defined in order to completely specify the behavior of SHOP2. The second subsection presents the formal semantics of operators and plans. The third subsection describes an assortment of functions within SHOP2 that are used to accomplish those semantics.

6.1 Internal Knowledge Structures

The following SHOP2 internal knowledge structures must be defined in order to fully specify the semantics of plan generation in SHOP2.

6.1.1 Substitutions

A **substitution** is a list of dotted pairs of the form

$$((x_1\ .\ t_1)\ (x_2\ .\ t_2)\ ...\ (x_k\ .\ t_k))$$

where every x_i is a variable symbol and every t_i is a term. If e is an expression and u is the above substitution, then the **substitution instance** e^u is the expression produced by starting with e and replacing each occurrence of each variable symbol x_i with the corresponding term t_i.

If d and e are two expressions, then:

- d is a **generalization** of e if e is a substitution instance of d;
- d is a **strict generalization** of e if d is a generalization of e but e is not a generalization of d;

- *d* and *e* are **equivalent** if each is a generalization of the other.

If *u* and *v* are two substitutions, then:

- *u* is a **generalization** of *v* if for every expression *e*, e^u is a generalization of e^v;
- *u* is a **strict generalization** of *v* if for every expression *e*, e^u is a strict generalization of e^v;
- *u* and *v* are **equivalent** if for every expression *e*, e^u and e^v are equivalent.

If *e* is an expression and x_1, x_2, ..., x_k are the variable symbols in *e*, then a **standardizer** for *e* is a substitution of the form

$$((x_1 \ . \ y_1) \ (x_2 \ . \ y_2) \ ... \ (x_k \ . \ y_k))$$

where each y_i is a new variable symbol that is not used anywhere else. Note that if *u* is a standardizer for *e*, then *e* and e^u are equivalent expressions.

If *d* and *e* are expressions and there is a substitution *u* such that $d^u = e^u$, then *d* and *e* are **unifiable** and *u* is a **unifier** for them. A unifier of *d* and *e* is a **most general unifier** (or **mgu**) of *d* and *e* if it is a generalization of every unifier of *d* and *e*. Note that all mgu's for *d* and *e* are equivalent.

6.1.2 States and Satisfiers

A **state** is a list of ground atoms intended to represent some "state of the world". A conjunct *C* is a **consequent** of a state *S* and an axiom list *X* if every logical expression *l* in *C* is a consequent of *S* and *X*. A logical expression *l* is a consequent of *S* and *X* if one of the following is true:

- *l* is an atom in *S*;
- *l* is a ground expression of the form `(eval p` t_1 t_2 … t_n`)`, and the evaluation of p with arguments $t_1, t_2, \ldots, t_n$ returns a non-`nil` value;
- *l* is a ground expression of the form `(call p` t_1 t_2 … t_n`)`, and the evaluation of p with arguments $t_1, t_2, \ldots, t_n$ returns a non-`nil` value;
- *l* is an expression of the form `(not` *a*`)`, and the atom *a* is not a consequent of *S* and *X*;
- *l* is an expression of the form `(assign` *v t*`)`, where *v* is a variable symbol and *t* is any Lisp expression. The value of *t*, which was evaluated via a call to the Lisp evaluator, is a substitution of *v*, i.e.*(v . t)*. This term is always a consequent of *S* and *X*;
- *l* is an expression of the form `(or` l_1 l_2 ... l_n`)`, where l_1, l_2, l_3, ..., l_n are logical expressions, and at least one expression in this list is a consequent of *S* and *X;*
- *l* is an expression of the form `(forall` *V Y Z*`)`, where *Y* and *Z* are logical expressions and *V* is the list of variables in *Y* such that for every satisfier *u* that satisfies *Y* in *S* and *X*, *u* also satisfies *Z* in *S* and *X*;
- *l* is an expression of the form `(imply` *Y Z*`)`, where *Y* and *Z* are logical expressions such that a satisfier *u* satisfies *Y* in *S* and *X* also satisfies *Z* in *S* and *X*;

- there exists a substitution v and an axiom `(:-` a `n1` C_1 `n2` C_2 ... n_n C_n`)` in X such that $l = a^v$ and one of the following holds:
    - C_1^v is a consequent of S and X;
    - C_1^v is not a consequent of S and X, but C_2^v is a consequent of S and X;
    - neither C_1^v nor C_2^v is a consequent of S and X, but C_3^v is a consequent of S and X;
    - ...;
    - none of C_1^v, C_2^v, C_3^v, ..., C_{n-1}^v is a consequent of S in X, but C_n^v is a consequent of S and X.

If C is a consequent of S and X, then it is a **most general consequent** of S and X if there is no strict generalization of C that is also a consequent of S and X.

Let S be a state, X be an axiom list, and C be an ordinary conjunct. If there is a substitution u such that C^u is a consequent of S and X, then we say that S and X **satisfy** C and that u is the **satisfier**. The satisfier u is a **most general satisfier** (or **mgs**) if there is no other satisfier that is a strict generalization of u. Note that C can have multiple non-equivalent mgs's. For example, suppose X contains the "walking distance" axiom given earlier, and S is the state

```
((weather-is good)
   (distance home convenience-store 1)
   (distance home supermarket 2))
```

Then for the conjunct `((walking-distance ?y))`, there are two mgs's from S and X: `((?y . convenience-store))` and `((?y . supermarket))`.

Let S be a state, X be an axiom list, and C = `(:first` C' `)` be a tagged conjunct. If S and X satisfy C', then the **most general satisfier** (or **mgs**) for C from S and X is the *first* mgs for C' that would be found by a left-to-right depth-first search. For example, if S and X are as in the previous example, then for the tagged conjunct `(:first (walking-distance ?y))`, the mgs from S and X is `((?y . convenience-store))`.

6.2 Formal Semantics

Recall that a plan is a list of operator invocations with costs and that an operator has an add list and a delete list. Informally, the meaning of the plan is that the specified operators are performed in sequence, incurring the specified costs. Similarly, the meaning of the operator is that the assertions in the add list are added to the state and the assertions in the delete list are removed from the state. The meaning of a method is that when the method’s precondition is satisfied, the task specified in the method’s head can be performed by performing each of the tasks specified in the method’s tail.

This subsection elaborates these informal notions, presenting detailed formal semantics of operators and plans. It is of particular use to anyone who has a SHOP2 domain and wishes to prove theorems (e.g., correctness, completeness, etc.) regarding plans generated in that domain.

6.2.1 Semantics of Operators

The intent of an operator is to specify that the task *h* can be accomplished at a cost of *c*, by modifying the current state of the world to remove every logical atom in *D* and add every logical atom in *A* if *P* is satisfied in the current state. In order to prevent plans from being ambiguous, there should be at most one operator for each primitive task symbol.

Let *S* be a state, *X* be the list of axioms, *L* be the list of protected conditions, *t* be a primitive task atom, and *o* be a planning operator whose head, precondition, delete list, add list, and cost are *h*, *P*, *D*, *A*, and *c*, respectively. Suppose that there is an mgu *u* for *t* and *h*, such that h^u is ground, that none of the ground atoms in D^u are in the list of protected conditions, and P^u is satisfied in S. Then we say that o^u is **applicable** to *t*, and that h^u is a **simple plan** for *t*. If *S* is a state, then the state and the protection list produced by executing o^u (or equivalently, h^u) in *S* and *L* is the new state:

$$(S', L') = \text{result}(S,L,h^u) = \text{result}(S,L,o^u) = (S - D^u) \cup A^u.$$

where *S'* and *L'* are obtained by modifying the current state of the world and the list of protected conditions as follows:

- remove every logical atom in D^u from the current state;
- remove every protection condition in D^u from the list of protected conditions;
- for every expression `(forall V Y Z)` in D^u and every satisfier *v* such that *S* and *X* satisfy Y^v, remove every logical atom in Z^u from the current state;
- for every expression `(forall V Y Z)` in D^u and every satisfier *v* such that *S* and *X* satisfy Y^v,, remove every protection condition in Z^u from the list of protected conditions;
- add every logical atom in A^u to the current state;
- add every protection condition in A^u to the list of protected conditions;
- for every expression `(forall V Y Z)` in A^u and every satisfier *v* such that *S* and *X* satisfy Y^v, add every logical atom in Z^u to the current state;
- for every expression `(forall V Y Z)` in A^u and every satisfier *v* such that *S* and *X* satisfy Y^v, add every protection condition in Z^u to the list of protected conditions.

Here is an example:

```
S           ((has-money john 40) (has-money mary 30))
T           (!set-money john 40 35)
O           (:operator (!set-money ?person ?old ?new)
               ((has-money ?person ?old))
               ((has-money ?person ?old))
               ((has-money ?person ?new)))
U           ((?person . john) (?old . 40) (?new . 35))
o^u         (:operator (!set-money john 40 35)
```

```
                    ((has-money john 40))
                    ((has-money john 40))
                    ((has-money john 35)))
```

h^u	`(!set-money john 40 35)`
Result(S,h^u)	`((has-money john 35) (has-money mary 30) )`
result(S,o^u)	`((has-money john 35) (has-money mary 30) )`

Here is an example using the forall keyword

S	`((location l1) (location l2) (location l3) (truck-at truck1 l1))`
T	`(!clear-locations)`
O	`(:operator (!clear-locations) ((forall (?l) ((location ?l) (not (truck-at ?t ?l))) ((location ?l)))) ())`
U	`()`
o^u	`(:operator (!clear-locations) ((forall (?l) ((location ?l) (not (truck-at ?t ?l))) ((location ?l)))) ())`
h^u =	`(!clear-locations)`
Result(S,h^u)	`((location l1) (truck-at truck1 l1))`
result(S,o^u)	`((location l1) (truck-at truck1 l1))`

6.2.2 Semantics of Methods

The purpose of a method is to specify the following:

- If the current state of the world satisfies C_1, then h can be accomplished by performing the tasks in T_1 in the order given;
- otherwise, if the current state of the world satisfies C_2, then h can be accomplished by performing the tasks in T_2 in the order given;
- ...;
- otherwise, if the current state of the world satisfies C_k, then h can be accomplished by performing the tasks in T_k in the order given.

Let S be a state, X be an axiom list, t be a task atom (which may or may not be ground), and m be the method `(:method` $h\ C_1\ T_1\ C_2\ T_2\ \ldots\ C_k\ T_k$`)`. Suppose there is an mgu u that unifies t with h; and suppose that m has a precondition C_i such that S and X satisfy C_i^u (if there is more than one such precondition, then let C_i be the first such precondition). Then we say that m is **applicable** to t in S and X, with the **active precondition** C_i and the **active tail** T_i. Then the result of applying m to t is the following set of task lists:

$$R = \{Call((T_i^u)^v)\text{: } v \text{ is an mgs for } C_i^u \text{ from } S \text{ and } X\}$$

where C*all* is SHOP2's evaluation function (the function that evaluates the values of the call-terms in the form `(call f t1 t2 .. tn)`). Each task list *r* in *R* is called a **simple reduction** of *t* by *m* in *S* and *X*. Here is an example:

S	`((has-money john 40) (has-money mary 30))`
X	`nil`
t	`(transfer-money john mary 5)`

M

```
(:method (transfer-money ?p1 ?p2 ?amount)
         ((has-money ?p1 ?m1)
          (has-money ?p2 ?m2)
          (call >= ?m1 ?amount))
         (:ordered (:task !set-money ?p1 ?m1
                                 (call - ?m1 ?amount))
                   (:task !set-money ?p2 ?m2
                                 (call + ?m2 ?amount))))
```

u

```
((?p1 . john) (?p2 . mary) (?amount . 5))
```

h^u

```
(transfer-money john mary 5)
```

C_1^u

```
((has-money john ?m1)
 (has-money mary ?m2)
 (call >= ?m1 5))
```

T_1^u

```
(:ordered (:task !set-money john ?m1 (call - ?m1 5))
          (:task !set-money mary ?m2 (call + ?m2 5))))
```

v

```
((?m1 . 40) (?m2 . 30))
```

$(C_1^u)^v$

```
((has-money john 40)
 (has-money mary 30)
 (call >= 40 30)))
```

$(T^u)^v$

```
(:ordered (:task !set-money john 40 (call - 40 5))
          (:task !set-money mary 30 (call + 30 5)))
```

$call((T^u)^v)$

```
(:ordered (:task !set-money john 40 35)
          (:task !set-money mary 30 35))
```

6.2.3 Semantics of Plans

Recall that a planning domain contains axioms, operators, and methods, and that a planning problem is a 4-tuple (S,M,L,D), where S is a state, M is a task list, L is a protection list, and D is a domain representation. Let T be the list of tasks in M that have no predecessor (i.e., those tasks can be performed at this time if they are applicable). If t is a task in T, and S is a state, then a **reduction** of t in S and D with respect to M and L that results in a new planning problem (S',M',L',D) is defined as follows:

> **if** t is a primitive task, **then**
>
> (S', L') = result(S,L,t);
>
> M' = the task list produced by removing t from M
>
> **else** t is a compound task, **then**

$S' = S$;
$L' = L$;
Suppose m is an applicable method to t in S, with unifier u, the active precondition C_i and the active tail T_i.
M' = the task list produced by replace t with T_i^u in M

endif

If $P = (p_1\ p_2\ \ldots\ p_n)$ is a plan, then we say that P **solves** (S,M,L,D), or equivalently, that P achieves M from S in D (we will omit the phrase "in D" if the identity of D is obvious) in any of the following cases:

Case 1.

both M and P are empty.

Case 2.

$T = (t_1\ t_2\ \ldots\ t_k)$ is a list of tasks in M that have no predecessor for which there is a task t_i that has the `:immediate` keyword and is applicable to the current state S. Let (S', M', L') = reduction(t_i, S, M, L). We say P **solves**(S,M,L,D) if either of the following is true.

- t_i is primitive and $p_1 = t_i$ and $(p_2\ p_3 \ldots p_n)$ solves (S', M', L', D)
- t_i is not primitive, and P solves (S', M', L', D)

Case 3.

$T = (t_1\ t_2\ \ldots\ t_k)$ is a list of tasks in M that have no predecessor, where t_i is a task in T that's applicable to the current state S. Let (S', M', L') = reduction(t_i, S, M, L). We say P **solves** (S,M,L,D) if either of the following is true.

- t_i is primitive and $p_1 = t_i$ and $(p_2\ p_3 \ldots p_n)$ solves (S', M', L', D)
- t_i is not primitive and P solves (S', M', L', D)

The planning problem (S,M,L,D) is **solvable** if there is a plan that solves it. For example, suppose that

S	`nil`
M	`((:ordered (:task do-both op1 op2)))`
T	`((:task do-both op1 op2))`
L	`nil`

D	`((:operator (!do ?operation) nil ((did ?operation)))`
	`(:method (do-both ?x ?y) nil`
	`(:ordered (:task !do ?x)`
	`(:task !do ?y)))`
	`(:method (do-both ?x ?y) nil`

	`(:ordered (:task !do ?y)` `(:task !do ?x))))`
P_1 P_2	`((do op1) 1 (do op2) 1))` `((do op2) 1 (do op1) 1))`

Then P_1 and P_2 are all of the plans that solve (*S*,*M*,*L*,*D*).

6.3 Key Functions in SHOP2

Below are some important functions in the Lisp implementation of SHOP2. They should be of interest to anyone who wishes to modify SHOP2 or to directly access internal capabilities of SHOP2. Pseudocode algorithms for the main planning functions of SHOP2 are also presented.

`(apply-substitution e u)`

e is an expression and *u* is a substitution. The function returns e^u.

`(compose-substitutions u v)`

If *u* and *v* are substitutions, then this function returns a substitution *w* such that for every expression *e*, $e^w = (e^u)^v$.

`(standardizer e)`

This function returns a standardizer for *e*.

`(standardize e)`

This function is equivalent to `(apply-substitution e (standardizer e))`.

`(unify d e)`

This procedure returns an mgu for the expressions *d* and *e* if they are unifiable, and returns `fail` otherwise.

`(find-satisfiers C S &optional just-one)`

If *C* is a conjunct and *S* is a state, then this function returns a list of mgs's, one for every most general instance of *C* that is satisfied by *S*. If the optional argument *just-one* is not `nil`, then the function returns the first mgs it finds, rather than all of them. Calling `(find-satisfiers C S)` is roughly equivalent to calling the following (simplified) pseudocode:

procedure find-satisfiers(C, S)
    if C is empty **then return** {nil}
    l = the first logical atom in C; B = the remaining logical atoms in C
    answers = nil
    if l is an expression of the form (not e) **then**
        if find-satisfiers(e, S, nil) = nil **then**
            return find-satisfiers(B, S)
        else
            return nil

```
        end if
else if l is an expression of the form (eval e) then
        if eval(e) is not nil then
                return find-satisfiers(B, S)
        else
                return nil
        end if
else if l is an expression of the form (or p1 p2 ... pn) then
        for every unifier u that unifies any pi with l
                for every v in find-satisfiers(B^u, S)
                        insert compose-substitutions(u,v) into answers
                end for
        end for
else if l is an expression of the form (imply C1 C2) then
        mgu = find-satisfiers(C1, S)
        if mgu is null or there exist a unifier u in mgu such that
                find-satisfiers(C2^u, S) is not equal to nil then
                return find-satisfiers(B, S)
        else
                return nil
        end if
else if l is an expression of the form
        (forall variables bounds conditions) then
        mgu = find-satisfiers(bounds, S)
        if mgu is null or for every unifier u in mgu,
                find-satisfiers(conditions^u, S) is not equal to nil then
                return find-satisfiers(B, S)
        else
                return nil
        end if
else
        for every atom s in S that unifies with l
                let u be the unifier
                for every v in find-satisfiers(B^u, S)
                        insert compose-substitutions(u,v) into answers
                end for
        end for
        for every axiom x in *axioms* whose head unifies with l
                let u be the unifier
                if tail(x) contains a conjunct D such that
                        find-satisfiers(append(D^u, B^u), S) is not nil then
                        let D be the first such conjunct
                        for every v in find-satisfiers(append(D^u, B^u), S)
                                insert compose-substitutions(u, v)
                                        into answers
                        end for
```

end if
end for
end if
return answers
end find-satisfiers

In this pseudo-code, *axioms* is an internal variable of SHOP2. It holds the list of the axioms defined for the domain under consideration.

`(apply-method S t m)`

If *S* is a state, *t* is a task, and *m* = `(:method h` C_1 T_1 C_2 T_2 ... C_k T_k`)` is a method, then this function does the following:

- If *m* is not applicable to *t* in *S*, then the function returns the symbol `FAIL`.
- If *m* is applicable to *t* in *S* and C_i is the active precondition, then the function returns a list of all simple reductions of T_i, one for each satisfier of C_i in *S*.

`(apply-operator S t o)`

If *S* is a state, *t* is a task, and *o* is an operator, then this function does the following:

- If there is an mgu *u* for *o* and *t*, then it returns the state produced by executing o^u in *S*.
- Otherwise, it returns `FAIL`.

```
(find-plans problem &key which verbose pshort gc pp state plan-tree
                         optimize-cost time-limit explanation)
```

This function implements the SHOP2 planning algorithm. For more about the arguments to and use of this function, see Section 5.1. A brief overview of the algorithm for this function is presented here. Calling `find-plans` is roughly equivalent to calling the following pseudocode, where *S* is the current state, *T* is a partially ordered set of tasks, and *L* is a list of protected conditions:

procedure find-plans (*S*,*T*,*L*)
if *T* is empty **then**
return NIL
endif
nondeterministically choose a task *t* in *T* that has no predecessors
<*r*,*R'*> = reduction (*S*,*t*)
if *r* = FAIL **then**
return FAIL
endif
nondeterministically choose an operator instance *o* applicable to *r* in *S*
S' = the state produced from *S* by applying *o* to *r*
L' = the protection list produced from *L* by applying *o* to *r*

```
        T' = the partially ordered set of tasks produced from T by replacing t
             with R'
        P = find-plans (S',T',L')
        return cons(o,P)
end find-plans


procedure reduction (S,t)
        if t is a primitive task then
             return <t,NIL>
        else if no method is applicable to t in S then
             return <FAIL,NIL>
        endif
        nondeterministically let m be any method applicable to t in S
        R = the decomposition (partially ordered set of tasks) produced by m
             from t
        r = any task in R that has no predecessors
        <r',R'> = reduction (S,r)
        if r' = FAIL then
                return <FAIL,NIL>
        endif
        R'' = the partially ordered set of tasks produced from R by replacing r
             with R'
        return <r',R''>
end reduction
```

`(defdomain domain-name D)`

This macro gives the name *domain-name* to planning domain *D*. (More specifically, what it does is to store *D*'s axioms, operators, and methods on *domain-name*'s property list.)

`(defproblem problem-name domain-name S T)`

This macro gives the name *problem-name* to the planning problem *(S,T,D)*, where *D* is the planning domain whose name is *domain-name*. (More specifically, what it does is to store *S*, *T*, and *domain-name* on *problem-name*'s property list.)

`(def-problem-set set-name list-of-problems)`

This macro gives the name *set-name* to the set of planning problems in *list-of-problems*. (More specifically, what it does is to store *list-of-problems* on *set-name*'s property list.)

Note that for backwards compatibility, SHOP2 also accepts the forms make-domain, make-problem, and make-problem-set, which were employed in SHOP 1.x, using the same arguments as defdomain, defproblem, and def-problem-set. The difference between the make-X and def-X forms is that in the latter case since the form itself is a macro, the arguments are not evaluated. This changes the syntax one uses. Thus in a SHOP 1.x domain one might define a problem as

(make-problem ‘problem-name ‘domain-name
‘(list of state atoms)
‘(list of tasks to be accomplished))

whereas in SHOP2 the syntax becomes

(defproblem problem-name domain-name
(list of state atoms)
(list of tasks to be accomplished))

where the arguments are all quoted in the SHOP 1.x make-problem function, they are unquoted when using the SHOP2 defproblem macro.

`(print-axioms &optional` *`name`*`)`

This function prints a list of the axioms for the domain whose name is *name*; defaults to the most recently defined domain.

`(print-operators &optional` *`name`*`)`

This function prints a list of the operators for the domain whose name is *name*; defaults to the most recently defined domain.

`(print-methods &optional` *`name`*`)`

This function prints a list of the methods for the domain whose name is *name*; defaults to the most recently defined domain.

`(get-state` *`name`*`)`

This function returns the initial state for the problem whose name is *name*.

`(get-tasks` *`name`*`)`

This function returns the list of tasks for the problem whose name is *name*.

`(get-problems` *`name`*`)`

This function returns the list of problem names for the problem set whose name is *name*.

`(do-problems name-or-list &rest keywords)`

name-or-list should be either a list of problem names or the name of a problem set. This function runs `find-plans` on each planning problem specified by the list or problem set, and then returns `nil`. The keywords are simply passed on to `find-plans`.

7 Java Interface for SHOP2

A new feature in SHOP2 version 1.1 is the Java interface. This interface is found in the `ji4shop2` (Java interface for SHOP2) directory within the SHOP2 distribution. The interface allows you to run SHOP2 from within a Java program. The interface spawns an external Lisp process in which SHOP2 is executed. It then retrieves the resulting plan from the Lisp process and makes it available to the user. Technical documentation on the packages and classes of the interface can be found in HTML in the `ji4shop2/doc` directory. A good starting point for examining that documentation is the `Shop` class, especially the two constructors for that class.

To compile the interface on a UNIX system or another system that has a make command, type *make*. Otherwise, just run *javac* on all of the java files. Before running the interface you will need to edit the config.txt file in the `ji4shop2` directory. You must enter the command you use to start Lisp, and a command line argument to load a Lisp file. Example arguments for several popular Lisp environments are provided. There are demo commands for UNIX (*demo*) and DOS/Windows (*demo.bat*). These will run the interface on very simple example files that are included. You may also want to try the interface with a verbosity level higher that 0, to see status information from SHOP2. To increase the verbosity level, add a number from 1 to 3 to the end of the command in the *demo* or *demo.bat* file. For details on the effects of specific verbosity numbers, see the discussion of the SHOP2 `:verbose` keyword in Section 5.1.

8 SHOP2 Graphical User Interface

The graphical user interface (GUI) for SHOP2 gives the user a visual representation of the planning process by dynamically displaying the decomposition tree. The GUI also keeps track of the methods that were used to decompose every task, the preconditions that had to be met prior to decomposition, and the current state of the world at each step of the planning process. The decomposition tree consists of yellow and blue nodes which represent ordered and unordered tasks respectively, and these nodes change in size from small to large as they are visited. Ordered nodes are displayed on-screen from top to bottom in accordance with their ordering constraint. For more help with using the GUI for SHOP2, please see its readme file.

9 Differences between SHOP 1.x and SHOP2

The differences between SHOP2 and SHOP 1.x can be grouped in two sets: syntactic changes in the domain definitions and differences in functionality.

9.1 SHOP 1.x Syntax Comparison

The table below gives examples of SHOP 1.x syntax, comparing it to the syntax used in SHOP2:

Previous syntax	**New Syntax**
(make-domain 'travel '((:- (have-taxi-fare ?distance) ((have-cash ?m) (eval (>= ?m (+ 1.5 ?distance))))) (:- (walking-distance ?u ?v) ((weather-is'good) (distance ?u ?v ?w) (eval (<= ?w 3))) ((distance ?u ?v ?w) (eval (<= ?w 0.5)))) (:method (pay-driver ?fare) ((have-cash ?m)	(defdomain travel ((:- (have-taxi-fare ?distance) ((have-cash ?m) (call >= ?m (call + 1.5 ?distance)))) (:- (walking-distance ?u ?v) good ((weather-is good) (distance ?u ?v ?w) (call <= ?w 3)) bad ((distance ?u ?v ?w) (call <= ?w 0.5))) (:method (pay-driver ?fare) ((have-cash ?m)

```
           (eval (>= ?m ?fare)))
         `((!set-cash ?m ,(- ?m ?fare))))

    (:method (travel-to ?q)
         ((at ?p) (walking-distance ?p ?q))
          '((!walk ?p ?q)))

    (:method (travel-to ?y)
       ((at ?x) (at-taxi-stand ?t ?x)
        (distance ?x ?y ?d) (have-taxi-fare ?d))
      `((!hail ?t ?x) (!ride ?t ?x ?y)
         (pay-driver ,(+ 1.50 ?d)))
       ((at ?x) (bus-route ?bus ?x ?y))
       `((!wait-for ?bus ?x) (pay-driver 1.00)
          (!ride ?bus ?x ?y)))

    (:operator (!hail ?vehicle ?location)
         ()
         ((at ?vehicle ?location)))

    (:operator (!wait-for ?bus ?location)
         ()
         ((at ?bus ?location)))

     (:operator (!ride ?vehicle ?a ?b)
        ((at ?a) (at ?vehicle ?a))
        ((at ?b) (at ?vehicle ?b)))

     (:operator (!set-cash ?old ?new)
        ((have-cash ?old))
        ((have-cash ?new)))

     (:operator (!walk ?here ?there)
        ((at ?here))
        ((at ?there)))
 ))
```

```
              (call >= ?m ?fare))
         ((!set-cash ?m ( call - ?m ?fare))))

    (:method (travel-to ?q)
         ((at ?p) (walking-distance ?p ?q))
          ((!walk ?p ?q)))

    (:method (travel-to ?y)
       by-taxi
       ((at ?x) (at-taxi-stand ?t ?x)
       (distance ?x ?y ?d) (have-taxi-fare ?d))
       ((!hail ?t ?x) (!ride ?t ?x ?y)
        (pay-driver (call + 1.50 ?d)))
       by-bus
      ((at ?x) (bus-route ?bus ?x ?y))
      ((!wait-for ?bus ?x) (pay-driver 1.00)
         (!ride ?bus ?x ?y)))

    (:operator (!hail ?vehicle ?location)
         () ()
         ((at ?vehicle ?location)))

    (:operator (!wait-for ?bus ?location)
         () ()
         ((at ?bus ?location)))

     (:operator (!ride ?vehicle ?a ?b)
        ()
        ((at ?a) (at ?vehicle ?a))
        ((at ?b) (at ?vehicle ?b)))

     (:operator (!set-cash ?old ?new)
        ((have-cash ?old))
        ((have-cash ?old))
        ((have-cash ?new)))

     (:operator (!walk ?here ?there)
        ((at ?here))
        ((at ?here))
        ((at ?there)))
 ))
```

```
(make-problem 'go-park-rich 'travel

   `((distance downtown park 2)
    (distance downtown uptown 8)
    (distance downtown suburb 12)
    (at-taxi-stand taxi1 downtown)
    (at-taxi-stand taxi2 downtown)
    (bus-route bus1 downtown park)
    (bus-route bus2 downtown  uptown)
    (bus-route bus3 downtown  suburb)
    (at downtown)
    (weather-is good)
    (have-cash 80))

'((travel-to park)))
```

```
(defproblem go-park-rich travel

   ((distance downtown park 2)
   (distance downtown uptown 8)
   (distance downtown suburb 12)
   (at-taxi-stand taxi1 downtown)
   (at-taxi-stand taxi2 downtown)
   (bus-route bus1 downtown park)
   (bus-route bus2 downtown uptown)
   (bus-route bus3 downtown  suburb)
   (at downtown)
   (weather-is good)
   (have-cash 80))

((travel-to park)))
```

(make-problem-set 'travel
    '(go-park-broke
       go-park-rich
 ))
	(def-problem-set travel
	    (go-park-broke
	      go-park-rich
	))
	
	*def-problem-set is not available in JSHOP

(do-problems 'travel :which :all)	Works in SHOP2 without any changes. Not available in JSHOP.

To summarize, the changes in syntax are as follows:

- Quotes, back quotes and commas are not used in SHOP2 except as needed in general Lisp expressions.
- *make-domain* is replaced with *defdomain.*
- *make-problem* is replaced with *defproblem.*
- Operators may have preconditions in SHOP2.
- The tail of an axiom can have names for each of the conjuncts, and the tail of a method can have names for each of the precondition-tail pairs. These names are optional.
- An ordinary list can be differentiated from a predicate or a function by inserting the optional label, *list*, before the first element of the list.
- In SHOP 1.x it was valid to have methods of the following form, where *e* is any Lisp expression:

```
(:method  (varSubtasks ?tasklist)
          ((precondition))
          e)
```

  This kind of method is not supported in SHOP2.

Note that in many cases, SHOP2 is able to process SHOP 1.x syntax (i.e., SHOP2 is partially backward compatible). For example, the old *make-domain* and *make-problem* forms can be handled by SHOP2. It is recommended, however, that new domains be written using the new SHOP2 forms.

9.2 SHOP 1.x Functionality Comparison

The following are some key differences between the functionality of SHOP 1.x and SHOP2:

- Unlike SHOP 1.x, SHOP2 allows the combination of partially ordered and fully ordered tasks through the use of the `:unordered` and `:ordered` keywords.
- The following keywords in SHOP2 domain definitions are not supported in SHOP 1.x: `or`, `assign`, `sort-by`, `forall`. The functionality associated with those keywords is not available in SHOP 1.x.

- The `not` keyword in SHOP 1.x can only be applied to individual atoms, not to arbitrary logical expressions.
- The `:optimize-cost`, `:time-limit`, and `:plan-tree` keyword arguments for SHOP2 are not supported in SHOP 1.x. The functionality associated with those keywords is not available in SHOP 1.x.
- The debugging features in SHOP2 (see Sections 5.2 and 5.3) are not present in SHOP 1.x.
- SHOP2 version 1.1 includes a Java interface. SHOP 1.x has no such interface.

SHOP2 is largely backward compatible with SHOP 1.x . SHOP2 can run most SHOP 1.x knowledge bases with only little or no modification.

10 Differences between SHOP2 and JSHOP 1.0

The following are key differences between SHOP2 and JSHOP 1.0:

- A **list-term** in SHOP2 is a term having the form `([list]` t_1 t_2 ... t_n`)`. The keyword *list* is not optional in JSHOP 1.0. In addition, JSHOP 1.0 also accepts the syntax (. t l) for lists. As this generates parsing errors from many Common Lisp readers it should be avoided in code that is intended to be portable between SHOP and JSHOP.
- A **call-term** is an expression of the form `(call f` t_1 t_2 ... t_n`)` where *f* is a function symbol and each t_i is a term or a call-term. JSHOP 1.0 restricts what functions to use for *f* in a call-term. We do not have such restrictions on *f* in SHOP2.
- The following keywords in SHOP2 domain definitions are not supported in JSHOP 1.0: `or`, `assign`, `sort-by`, `forall`, `:immediate`, `:protection`. The functionality associated with those keywords is not available in JSHOP 1.0.
- The `not` keyword in JSHOP 1.0 can only be applied to individual atoms, not to arbitrary logical expressions.
- The `:optimize-cost` and `:time-limit` keyword arguments for SHOP2 are not supported in JSHOP 1.0. The functionality associated with those keywords is not available in JSHOP 1.0.
- SHOP2 optionally allows the use of an iterative-deepening search for a plan.
- SHOP2 can print out statistics such as the depth and cost of the plan.
- In JSHOP 1.0, the depth first search for the plan is not allowed to have a depth limit.
- JSHOP 1.0 does not support partially ordered tasks (i.e., the `:ordered` and `:unordered` keywords in task lists).
- The debugging features in SHOP2 (see Section 6) are not present in JSHOP 1.0
- Once SHOP2 is compiled and loaded into the Lisp environment, the form *defdomain* can be used to define a planning domain. Once a domain is defined, the forms *defproblem* and *def-problem-set* are used to define planning problems and groups of planning problems, respectively. Then the form *find-plans* is used to run SHOP2 over a single problem, or *do-problems* over a problem set. *defproblem, def-problem-set, do-problem* and *do-problem-set* are not

implemented in JSHOP 1.0. Instead JSHOP 1.0 parses and solves all the problems in a problem file.

- When JSHOP 1.0 finds a plan, it also returns the following items that SHOP2 does not return:
  - The final state that will be reached upon applying the plan,
  - The list of state atoms that will be added and deleted from current state to reach the final state,
  - The derivation tree for the plan (SHOP2 will return this only if the *plan-tree* keyword is used in invoking it).

11 General Notes on SHOP2

1. Since the null conjunct is always true, an axiom of the form `(:- a nil)` is equivalent to asserting the atom *a* as a basic fact. The difference is that the expression `(:- a nil)` is what one would put into the set of axioms for the problem domain, whereas the atom *a* is what one would put into a state description. An atom *a* in the state description can be deleted by an operator. However, if we have an axiom `(:- a nil)`, then *a* is always true, no operator can change that.

2. An axiom with several conjuncts in its tail has a different semantics than what you would get by making each conjunct the tail of a separate axiom. For example, consider the following axiom lists:

```
X1 = ((:- (a ?x) ((b ?x))
                 ((c ?x)))))

X2 = ((:- (a ?x) ((b ?x)))
      (:- (a ?x) ((c ?x))))
```

   In X_1, the single axiom acts like an *if-then-else*: if ((b ?x)) is true then `find-satisfiers` returns the satisfiers for (b ?x); otherwise if ((c ?x)) is true then it returns the satisfiers for (c ?x). For example,

```
(find-satisfiers '((a ?u)) '((b 2) (c 3)))
```

   would return

```
(((?u . 2))).
```

   On the other hand, in X_2, the set of axioms acts like a logical "or": `find-satisfiers` returns every satisfier for (b ?x) and every satisfier for (c ?x). In this case,

```
(find-satisfiers '((a ?u)) '((b 2) (c 3)))
```

   `would return`

```
(((?u . 2)) ((?u . 3))).
```

3. Since a primitive task name is basically a call to an operator, you should never create a set of methods and operators that has more than one operator for the same primitive task. Otherwise, your plans will be ambiguous.

4. The following two calls to `find-plans` SHOP2 will find the same set of all shallowest plans, but in the first case SHOP2 will use a depth-first search and in the second case it will use an iterative-deepening search:

```
(find-plans 'p :which :all-shallowest)
(find-plans 'p :which :id-all)
```

   Likewise, the following two calls to SHOP2 will both find the same shallowest plan, but in the first case SHOP2 will use a depth-first search and in the second case it will use an iterative-deepening search:

```
(find-plans 'p :which :shallowest)
(find-plans 'p :which :id-first)
```

12 Acknowledgments

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F30602-99-1-0013 and F30602-00-2-0505, Army Research Laboratory DAAL01-97-K0135, and National Science Foundation DMI-9713718, and the University of Maryland General Research Board. Opinions expressed here do not necessarily reflect those of the funders.

13 References

[Dean and Boddy, 1998] T. Dean and M. Boddy. An analysis of time-dependent planning. In *AAAI-88*, 1988.

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI-99*, 1999.

[Nau *et al.*, 2000] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP and M-SHOP: Planning with Ordered Task Decomposition. Tech report TR 4157, University of Maryland, College Park, MD, June 2000.

[Nau *et al.*, 2001] D. S. Nau, H. Muñoz-Avila Y. Cao, A. Lotem, and.S. Mitchell. Totally Ordered Planning with Partially Ordered Subtasks. In *IJCAI-01*, 2001.