Documentation for JSHOP2

Okhtay Ilghami
Department of Computer Science
University of Maryland
College Park, MD 20742
USA
okhtay@cs.umd.edu

Technical Report CS-TR-4694

May 9, 2006

Contents

1 Introduction **2**

2 Notations Used in This Document **2**

3 The JSHOP2 Formalism **3**
- 3.1 Symbols . . . 3
- 3.2 Terms . . . 4
  - 3.2.1 List Terms . . . 4
  - 3.2.2 Call Terms . . . 4
- 3.3 Logical Atoms . . . 5
- 3.4 Logical Expressions . . . 5
  - 3.4.1 Conjunctions . . . 5
  - 3.4.2 Disjunctions . . . 5
  - 3.4.3 Negations . . . 5
  - 3.4.4 Implications . . . 6
  - 3.4.5 Universal Quantifications . . . 6
  - 3.4.6 Assignments . . . 6
  - 3.4.7 Call Expressions . . . 7
- 3.5 Logical Precondition . . . 7
  - 3.5.1 First Satisfier Precondition . . . 7
  - 3.5.2 Sorted Precondition . . . 7
- 3.6 Axioms . . . 8
- 3.7 Task Atoms . . . 8
- 3.8 Task Lists . . . 8
- 3.9 Operators . . . 9
- 3.10 Methods . . . 11
- 3.11 Planning Domain . . . 12
- 3.12 Planning Problem . . . 12
- 3.13 Plans . . . 12
- 3.14 Miscellaneous Points . . . 13

4 Plan Generation Process **14**

5 Internal Technical Information **17**
- 5.1 Internal Knowledge Structures . . . 17
  - 5.1.1 Substitutions . . . 17
  - 5.1.2 States and Satisfiers . . . 18
- 5.2 Formal Semantics . . . 20
  - 5.2.1 Semantics of Operators . . . 20
  - 5.2.2 Semantics of Methods . . . 21
  - 5.2.3 Semantics of Plans . . . 25

A Implementing and Calling External Functions **27**

B Implementing Comparator Functions to Be Used with Sorted Preconditions **27**

C Syntax Differences with SHOP2 **28**

1 Introduction

This document presents the design and implementation details of JSHOP2. JSHOP2 is the Java implementation of SHOP2 (Simple Hierarchical Ordered Planner). The algorithms of SHOP2 and its predecessor SHOP have originally been designed by Professor Dana S. Nau, and implemented, maintained and updated by the entire SHOP research group. For more information on SHOP and SHOP2 algorithms see [2, 3].

Significant portions of this document are copied from the previous SHOP and SHOP2 documentations with minor modifications and adjustments to reflect the Java implementation. Previous documentations are included in the SHOP and SHOP2 distributions, and are authored and maintained by the entire SHOP team.

JSHOP2 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. JSHOP2 has the following characteristics:

- JSHOP2 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.
- JSHOP2 can be used to create very efficient domain-specific planning algorithms. The JSHOP2 software distribution includes several examples of such domain algorithms.
- JSHOP2 incorporates many features from PDDL, e.g., support for quantifiers and conditional effects in methods and operators.
- JSHOP2 allows the combination of partially ordered tasks through the use of the :unordered keyword.

2 Notations Used in This Document

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

- **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.

- Typewriter characters are used to write computer code, or refer to the variables in the code. For example:

  (call <= 7 (call + 5 3))

- *Italic* characters refer to special words or symbols. For example:

  Let a be a *logical atom*.

- 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] L_1 [name_2] L_2 ... [name_n] L_n)

3 The JSHOP2 Formalism

The inputs to JSHOP2 are a *planning domain* and a *planning problem*. Planning domains are composed of *operators*, *methods*, and *axioms*. Planning problems are composed of *logical atoms* (an initial state) and *task lists* (high-level actions to perform).

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 *term*s. 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.

3.1 Symbols

In the structures defined below, there are five kinds of symbols: *variable symbols*, *constant symbols*, *primitive task symbols*, *compound task symbols*, and *function symbols*. Each symbol can consist of only letters, digits, question marks, exclamation points, hyphens and underlines. To distinguish among these symbols, JSHOP2 uses the following conventions:

- A **variable symbol** can be any symbol whose name begins with a question mark (such as ?x or ?hello-there).

- A **primitive task symbol** can be any symbol whose name begins with an exclamation point (such as !unstack or !putdown).
- A **constant symbol**, a **predicate symbol**, or a **compound task symbol** can be any symbol whose name begins with a letter or an underline.
- A **function symbol** can be any valid Java identifier.

Note that JSHOP2 keywords can not be used as symbols.

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

3.2 Terms

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

- A variable symbol.
- A constant symbol.
- A number.
- A *list term*.
- A *call term*.

3.2.1 List Terms

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

$$(t_1\ t_2\ \ldots\ t_n\ [.\ L])$$

where each t_i is a term. This specifies that t_1, t_2 ..., and t_n are the items of a list. The last optional element is added for backward compatibility with SHOP and SHOP2 (which are written in LISP), and represents the cdr part of a *cons cell*. If you do not know LISP, suffice to remember that the last parameter should usually evaluate to a list, and in such case it will represent the *rest* of the list after t_n.

3.2.2 Call Terms

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

$$(\text{call}\ f\ t_1\ t_2\ \ldots\ t_n)$$

where f is either a function symbol or a built-in JSHOP2 function (such as +) and each t_i is a term. A call-term has a special meaning to JSHOP2, because it tells JSHOP2 that f is an attached procedure, i.e., that whenever JSHOP2 needs to evaluate any structure where a call term appears, JSHOP2 should replace the call term with the result of applying the function f on the arguments t_1, t_2, ..., and t_n. For example, the following call term would have the value 6:

(call + (call + 1 2) 3)

JSHOP2 replaces a call term with its equivalent as soon as it is possible to evaluate the call term during the planning process (i.e., as soon as the call term becomes ground as a result of binding its variables).

Some of the most common functions are built into the JSHOP2 code, but this does not restrict the users to define whatever function they need and then call those functions within the input of JSHOP2. For more information on how to write external functions for JSHOP2 and call them in a JSHOP2 domain, see Appendix A.

3.3 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.

3.4 Logical Expressions

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

3.4.1 Conjunctions

A **conjunction** 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 (), or equivalently, nil) then the form always evaluates to true.

3.4.2 Disjunctions

A **disjunction** has the form

(or L_1 L_2 ... L_n)

where each L_i is a logical expression.

3.4.3 Negations

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

(not L)

where L is a logical expression.

```
(:operator (!drive ?t ?c1 ?c2)
   (forall (?p) (package ?p) (in ?p ?t))
   ((at ?t ?c1))
   ((at ?t ?c2))
)
```

Figure 1: Example of usage of the forall keyword in a logical expression.

3.4.4 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 when this logical expression is being evaluated, or the semantics of the implication will be ambiguous.

3.4.5 Universal Quantifications

A **universal quantification** expression 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. The intent of a universal quantification expression is to satisfy that, for each possible substitution u for variables in V, if Y^u is satisfied then Z^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 Subsection 3.9); the latter is used to express a set of effects rather than a logical expression and consequently has a different syntax. Figure 1 shows an example of usage of the forall keyword in a logical expression. The operator !drive which is used to represent driving a truck from one city to another, first checks if all the packages are loaded into the truck as its precondition.

3.4.6 Assignments

An **assignment expression** has the form

(assign v t)

where v is a variable symbol and t is a term. The intent of an assignment expression is to bind the value of t to the variable symbol v.

3.4.7 Call Expressions

A **call expression** has the same form as a call term, but semantically, its value is interpreted as true (anything but an empty list) or false (an empty list, or equivalently nil).

3.5 Logical Precondition

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

3.5.1 First Satisfier Precondition

A **first satisfier precondition** has the form

(:first L)

where L is a logical expression. Such a precondition causes JSHOP2 to consider only the first set of bindings that satisfies L. Alternative bindings will not be considered even if the first bindings found do not lead to a valid plan.

3.5.2 Sorted Precondition

A **sorted precondition** has the form

(:sort-by v [f] L)

where v is a variable symbol, f is the name of a class that implements Java's Comparator interface, or a built-in comparison function (there are two such functions, $<$ and $>$), and L is a logical expression. Such a precondition causes JSHOP2 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 JSHOP2 to consider bindings in decreasing (high to low) order of the value of ?d. If the comparison function f is omitted, it defaults to $<$, indicating increasing (low to high) order.

For more information on how to write user-defined comparator functions for JSHOP2 and call them in a JSHOP2 domain, see Appendix B.

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

Figure 2: A sample axiom.

3.6 Axioms

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

(:- a [name_1] L_1 [name_2] L_2 ... [name_n] L_n)

where the axiom's **head** is the logical atom a, and its **tail** is the list [name_1] L_1 [name_2] L_2 ... [name_n] L_n, and each L_i is a logical precondition and each name_i is a symbol called the **name** of L_i. The names of the expressions are optional. When a domain definition is loaded into JSHOP2, a unique name will be generated for each branch if no name was given. These names have no semantic meaning to JSHOP2, but are provided to help the user debug domain descriptions. The intended meaning of an axiom is that a is true if L_1 is true, or if L_1 is false but L_2 is true, ..., or if all of L_1, L_2, ... L_{n-1} are false but L_n is true. For example, the axiom in Figure 2 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.

3.7 Task Atoms

A **task atom** is an expression of the form

([:immediate] s t_1 t_2 ... t_n)

where s is a task symbol and the arguments t_1, t_2, ..., and 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. A task atom without an :immediate keyword is called an **ordinary** task atom while a task atom with that keyword 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.

3.8 Task Lists

A **task list** is either a task atom or an expression of the form:

([:unordered] [tasklist_1 tasklist_2 ... tasklist_n])

where $\mathsf{tasklist}_1$, $\mathsf{tasklist}_2$, ..., and $\mathsf{tasklist}_n$ are task lists themselves. Note that here, n can be zero, resulting in an empty task list.

If there is no :unordered keyword in a task list, it means that JSHOP2 must perform the task lists in the order that they are given. The :unordered keyword specifies that there is no particular ordering specified between $\mathsf{tasklist}_1$, $\mathsf{tasklist}_2$, ..., and $\mathsf{tasklist}_n$. With the use of the :unordered keyword, JSHOP2 may interleave tasks between different task lists. Suppose we have two task lists as the following:

$T = (t_1\ t_2\ \ldots\ t_m)$
$U = (u_1\ u_2\ \ldots\ u_n)$

and that we have the main task list

$M = (\text{:unordered}\ T\ U)$

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 JSHOP2 to interleave the tasks of T and the tasks of U. However, if some of the tasks are immediate, then each time JSHOP2 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 JSHOP2 should perform t_1 before both t_2 and u_1.

Note that 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 at any given point during the planning process. Otherwise, JSHOP2's behavior on those tasks is undefined. For instance, in the above example, t_1 and u_1 cannot both have the :immediate keyword.

3.9 Operators

An **operator** has the following form:

(:operator h P D A [c])

where:

- h (the operator's **head**) is a primitive ordinary task atom (i.e., a non-immediate task atom with a task symbol that begins with an exclamation point).
- P (the operator's **precondition**) is a logical precondition. P can contain any variable symbols that can be either in h or not, provided that there can be at most one satisfier for P in the current state at any given point in the planning process.
- D (the operator's **delete list**) is a list each of the elements of which may be any of the following:

```
(: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))
)
```

Figure 3: Sample operators that use protection conditions.

  - a logical atom that can contain only variable symbols that appear either in h or in P.
  - a *protection condition* that can contain only variable symbols that appear either in h or in P.
  - an expression of the form (forall V Y Z), where V is a list of variables in Y, Y is a logical expression, and Z is a list of logical atoms that contain no variable symbols other than those in h, P, or V.

- A (the operator's **add list**) is a list of elements that have the same form as the elements of D.
- c (the operator's **cost**) is a term. If c is omitted, its default value is 1.

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 JSHOP2 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 operators in Figure 3. The :immediate keyword is particularly useful to enforce ordering constraints when using unordered task lists.

Since several operators might want to protect the same atom, the way protection conditions are implemented is that each protected atom has a counter

associated with it, and every time an operator has that particular protection condition in its add list, that counter is increased by one, and whenever an operator has that protection condition in its delete list, the counter is decreased by one. An atom can not be deleted by an operator unless its associated protection counter is zero.

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 JSHOP2; 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 and the semantics for internal operators are identical to those of other operators. JSHOP2 handles internal operators exactly the same way as ordinary operators during planning. JSHOP2 includes these operators in any plans that it returns at the end of execution. The primary reason that the internal operator syntax exists in JSHOP2 is so that automated systems which use JSHOP2 plans as an input can easily distinguish between those operators which involve action and those which were merely internal to the planning process.

3.10 Methods

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

$$(\text{:method } h\ [\text{name}_1]\ L_1\ T_1\ [\text{name}_2]\ L_2\ T_2\ \ldots\ [\text{name}_n]\ L_n\ T_n)$$

where:

- h (the method's **head**) is a compound ordinary task atom (i.e., a non-immediate task atom that starts with a compound task symbol).
- Each L_i (a **precondition** for the method) is a logical precondition.
- Each T_i (a **tail** of the method) is a task list.
- Each name_i is the name for the succeeding (L_i T_i) pair. These names are optional and if omitted a unique name will be assigned for each pair. These names have no semantic meaning to JSHOP2, but are provided in order to help the user debug domain descriptions.

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 method's tails when 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 can not be satisfied. If there are multiple methods for a given

```
(:method (eat ?food)
  branch1
  (have-fork ?fork)
  ((!eat-with-fork ?food ?fork))
  branch2
  (have-spoon ?spoon)
  ((!eat-with-spoon ?food ?spoon))
)
```

```
(: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))
)
```

Figure 4: The code on the left hand side is semantically equivalent to the code on the right hand side.

task available at some point in time, all of these methods may be considered regardless of order.

Figure 4 shows an example of two semantically equivalent ways to write methods. In both the code on the left hand side and the code on the right hand side, the !eat-with-spoon operator may be performed only if the (have-spoon ?spoon) is satisfied and (have-fork ?fork) is not satisfied.

3.11 Planning Domain

A **planning domain** has the form

(defdomain domain-name (d_1 d_2 ... d_n))

where domain-name is a symbol and each item d_j is one of the following: an operator, a method, or an axiom.

3.12 Planning Problem

A **planning problem** has the form

(defproblem problem-name domain-name ([$a_{1,1}$ $a_{1,2}$ ... $a_{1,n}$]) T_1 ... ([$a_{m,1}$ $a_{m,2}$ ... $a_{m,n}$]) T_m)

where problem-name and domain-name are symbols, each $a_{i,j}$ is a ground logical atom, and each T_i is a task list. This form defines m planning problems in domain domain-name each of which may be solved by addressing the tasks in T_i with the initial state defined by the atoms $a_{i,1}$ through $a_{i,n}$.

3.13 Plans

A plan is a list of heads of ground instances of the operators in a given domain. If $p = (h_1\ h_2\ \ldots\ h_n)$ is a plan and S is a state, then p(S) is the state produced

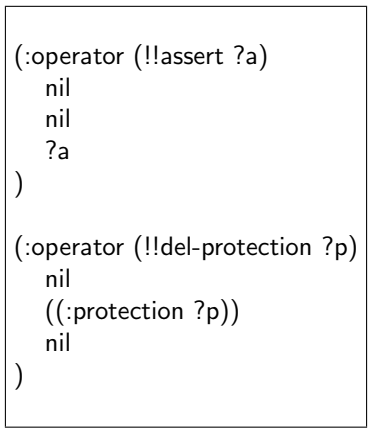

Figure 5: Examples of on-the-fly creation of JSHOP2 structures.

by starting with S and executing o_1, o_2, ..., and o_n in the order given. The **cost** of the plan p is the total cost of each of o_i's.

3.14 Miscellaneous Points

- JSHOP2 input (i.e., planning domains and planning problems) is generally not case-sensitive. The only exception to this rule is discussed in Appendix A.
- The reserved word nil can replace an empty list anywhere in the JSHOP2 input that an empty list is allowed.
- Spaces, tabs, and enters are ignored by JSHOP2. Also, anything in a line that follows a ";" is considered to be comment and is ignored.
- **On-the-fly** creation of two JSHOP2 structures is allowed, meaning that these structures can be simply defined as variable symbols in the input, but as planning process proceeds, these variables are supposed to be mapped to values that represent those structures. These two structures are logical atoms and operator delete or add lists. Figure 5 shows an example. There, variable symbol ?a is supposed to be later (before the application of the operator, of course) mapped to a list of logical atoms that represents the items in the !!assert operator's add list. The variable symbol ?p is supposed to be mapped to a logical atom, the one that the !!del-protection operator is going to unprotect.

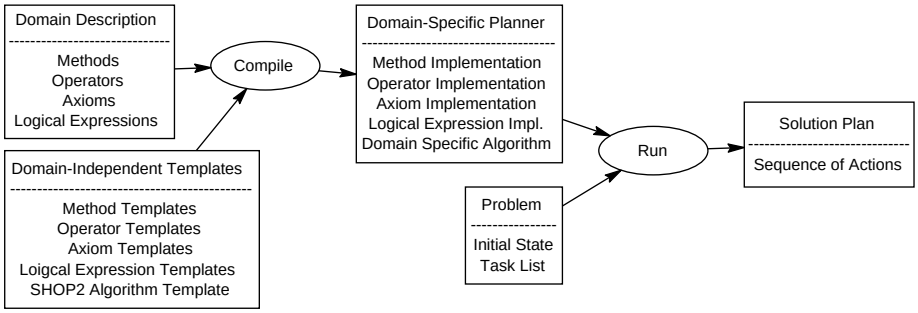

Figure 6: Compilation Process

4 Plan Generation Process

So far, most of the existing hand-tailorable planners (general-purpose planners that allow the users to feed them with domain-specific information on how to look for a plan) have been *interpreters* of their input (i.e., a general purpose program that can interpret different inputs and act accordingly). JSHOP2, however, is a *compiler* of its domain description, meaning that it compiles a given domain description to a domain-specific planner, and then runs that planner to solve the planning problems in that domain. Figure 6 shows this process. There are two reasons why we have chosen to implement JSHOP2 this way:

- There are certain optimizations that can be done only by taking advantage of the information in the domain description and producing code tailored for that particular domain description, rather than using a general purpose approach. For example, to handle a list that can have different sizes in different domain descriptions in a general purpose piece of code, we will need a list data structure. However, if we compile a domain specific piece of code for a domain where we know this list has size 2, then we can use an array of size 2 in the code instead, making the code more efficient. For more technical details about this approach, see [1].

- It is easier this way to implement external code calls and comparator functions in a Java implementation of SHOP2, given how rigorously-typed Java is (As opposed to LISP, the language SHOP2 was originally implemented in, where data and code are essentially the same thing, and therefore it is easier to have code calls in SHOP2's data structures).

Therefore, each class in the JSHOP2 source can be a compile-time class (i.e., a class that JSHOP2 uses at compile time to keep track of its data structures), a run-time class (i.e., a class that is used at run time to find plans), or both. Figure 7 shows the relationship between these two groups of classes. For more information on the internals of JSHOP2 classes see the HTML documentation in the JSHOP2 distribution (doc/index.html).

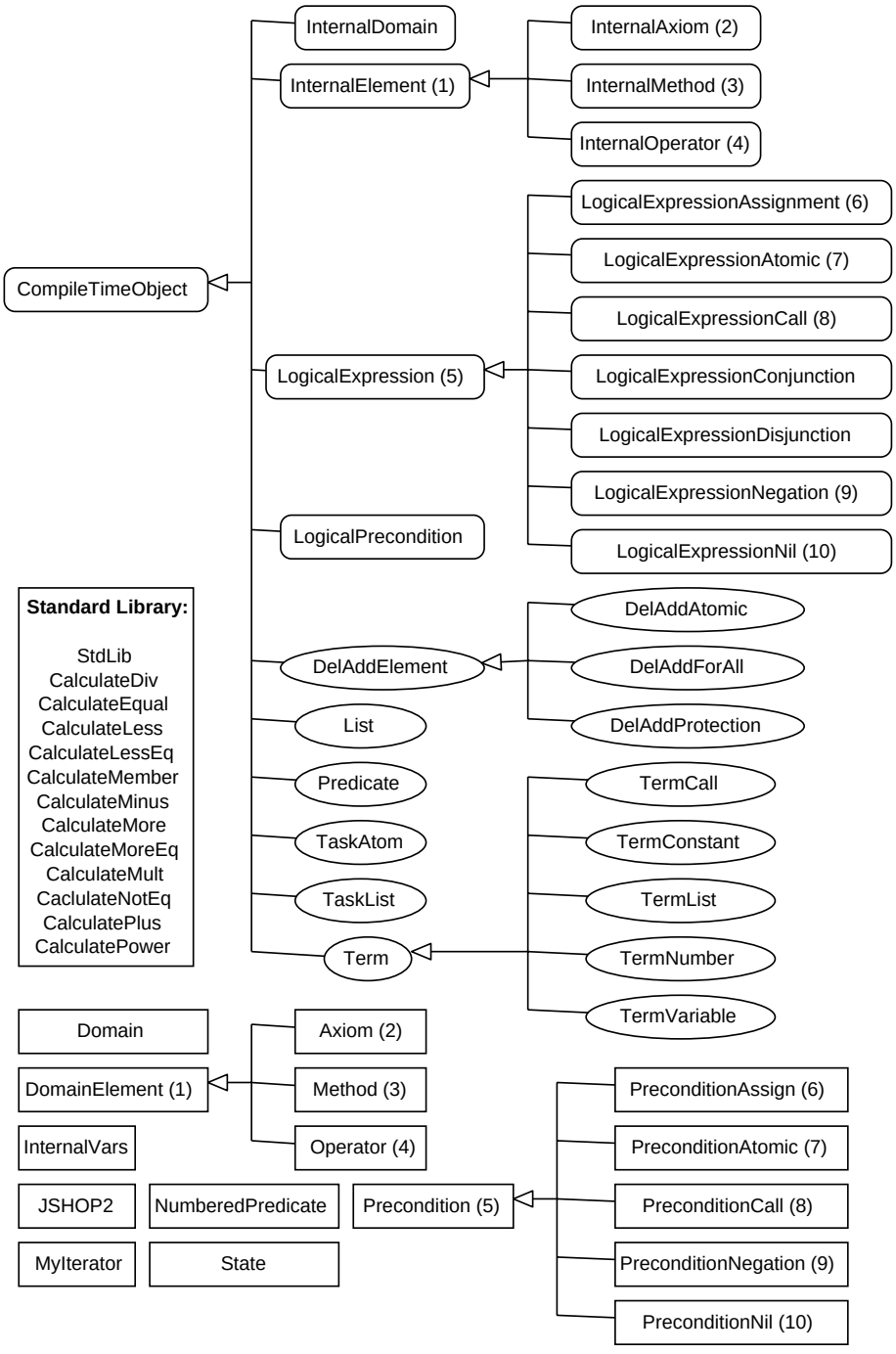

Figure 7: The class hierarchy in JSHOP2. Rectangles represent run time classes, rectangular ovals represent compile time classes, and ovals represent classes that are used both at compile and run time. Compile and run time classes that have the same number are corresponding to eachother.

To compile domain descriptions into executable Java code, JSHOP2 uses ANTLR software[1]. The JSHOP2 grammar is fed to ANTLR, which in turn compiles the domain and problem descriptions written in that grammar into Java code (The grammar can be found in src/JSHOP2/JSHOP2.g).

Solving a planning problem in JSHOP2 is done in three steps:

- First, the domain description should be compiled into Java code. This is done by this command-line command:

  java JSHOP2.InternalDomain *InputFileName*

  The result of this command will be a Java file (named after the domain name in the input domain description) that contains a class (with the same name) that implements the functionality of the input JSHOP2 domain.

- Second, the problem descriptions should be compiled into Java code. This is done by one of these three command-line commands:

  java JSHOP2.InternalDomain -r *InputFileName*
  Or
  java JSHOP2.InternalDomain -r*SomeInteger* *InputFileName*
  Or
  java JSHOP2.InternalDomain -ra *InputFileName*

  The result of these commands will be a Java file (named after the problem name in the input problem description) that contains a class (with the same name) that implements the functionality of the input JSHOP2 planning problems. The first of the above three commands creates a class that returns only the first plan found for each of the input planning problems, the second one returns at most *SomeInteger* plans for each problem, and the last one returns *all* of the valid plans for the input planning problems. Note that in the last case, it is the user's responsibility to make sure that there are only a tractable number of solution plans for each of the input planning problems.

- Third, the Java file produced in the previous step should be compiled and run. When this file is run, it will find the plan(s) for the input planning problem(s) and print it/them.

For more information on how to install, compile, and run JSHOP2, see the README file in the JSHOP2 distribution.

[1] See www.antlr.org for more information.

5 Internal Technical Information

This section presents information about the internal workings of the JSHOP2 planning process. Note that this section is primarily of interest to planning researchers and planning system developers. Most JSHOP2 users (especially beginning users) are advised to skip this section.

5.1 Internal Knowledge Structures

The following JSHOP2 internal knowledge structures must be defined in order to fully specify the semantics of plan generation in JSHOP2.

5.1.1 Substitutions

A **substitution** is an array with one element for each variable symbol in the current scope.[2] Each element corresponding to any given such variable is NULL if the variable is not bound to anything by this substitution, or is the term to which that variable is mapped.

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

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 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.

[2] The **scope** of the variables in JSHOP2 is limited to the method, axiom or operator they appear in, meaning that ?x in a method is different from ?x in another method or in an axiom or in an operator or in another instance of the same method.

5.1.2 States and Satisfiers

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

- L is an atom in S.
- There exists a substitution v and an axiom

  (:- a [name_1] L_1 [name_2] L_2 ... [name_n] L_n)

  in X such that $\mathsf{L} = \mathsf{a}^{\mathsf{v}}$; and one of the following holds:
  - $\mathsf{L}_1^{\mathsf{v}}$ is a consequent of S and X.
  - $\mathsf{L}_1^{\mathsf{v}}$ is not a consequent of S and X, but $\mathsf{L}_2^{\mathsf{v}}$ is a consequent of S and X.
  - ...
  - None of $\mathsf{L}_1^{\mathsf{v}}$, $\mathsf{L}_2^{\mathsf{v}}$, ..., or $\mathsf{L}_{n-1}^{\mathsf{v}}$ is a consequent of S and X, but $\mathsf{L}_n^{\mathsf{v}}$ is a consequent of S and X.
- 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, ..., and t_n returns a non-nil (i.e., anything other than an empty list) value.
- L is an expression of the form

  (not L')

  and the logical expression L' 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 a term. The value of t is a substitution of v. This term is always a consequent of S and X.
- L is an expression of the form

  ([and] [L_1 L_2 ... L_n])

  where L_1, L_2, ..., and L_n are logical expressions such that each L_i is a consequents of S and X. Note that by definition, an empty logical expression is a consequent of any state and any axiom list.
- L is an expression of the form

(or L_1 L_2 ... L_n)

where L_1, L_2, ..., and 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 if Y is a consequent of S and X then Z is also a consequent of S and X.

If L 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 L that is also a consequent of S and X.

Let S be a state, X be an axiom list, and L be a logical expression. If there is a substitution u such that L^u is a consequent of S and X, then we say that S and X **satisfy** L 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 L can have multiple nonequivalent 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 atom (walking-distance ?y), there are two mgs's from S and X: One that maps ?y to convenience-store, and one that maps ?y to supermarket.

Let S be a state, X be an axiom list, and L = (:first L') be a first satisfier precondition. If S and X satisfy L', then the most general satisfier (or mgs) for L from S and X is the first mgs for L' 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 first satisfier precondition (:first (walking-distance ?y)), the mgs from S and X is the one that maps ?y to convenience-store.

5.2 Formal Semantics

Recall that a plan is a list of operator invocations 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. 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 JSHOP2 domain and wishes to prove theorems (e.g., correctness, completeness, etc.) regarding plans generated in that domain.

5.2.1 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.

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 P^u is satisfied in S and X with mgs v (There should be at most one such mgs, or otherwise the semantics of the operator will be ambiguous), and none of the ground atoms in $(D^u)^v$ are in the list of protected conditions. Then we say that $(o^u)^v$ is **applicable** to t, and that $(h^u)^v$ is a **simple plan** for t. If S is a state, then the state and the protection list produced by executing $(o^u)^v$ (or equivalently, $(h^u)^v$) in S and L is the new state and protected condition list:

$$(S', L') = \text{result}(S, L, (h^u)^v) = \text{result}(S, L, (o^u)^v)$$

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 $(D^u)^v$ from the current state.
- Decrease the protection counter for every protection condition in $(D^u)^v$.
- For every expression

  (forall V Y Z)

  in $(D^u)^v$ and every satisfier w such that S and X satisfy Y^w, remove every logical atom in Z^w from the current state.
- For every expression

(forall V Y Z)

in $(D^u)^v$ and every satisfier w such that S and X satisfy Y^w, decrease the protection counter for every protection condition in Z^w.

- Add every logical atom in $(A^u)^v$ to the current state.
- Increase the protection counter for every protection condition in $(A^u)^v$.
- For every expression

  (forall V Y Z)

  in $(A^u)^v$ and every satisfier w such that S and X satisfy Y^w, add every logical atom in Z^w to the current state.
- For every expression

  (forall V Y Z)

  in $(A^u)^v$ and every satisfier w such that S and X satisfy Y^w, increase the protection counter for every protection condition in Z^w.

Figure 8 shows an example of an operator being applied. Figure 9 shows another example of an operator that uses the forall keyword.

5.2.2 Semantics of Methods

The purpose of a method is to specify the following:

- If the current state of the world satisfies L_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 L_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 L_n, then h can be accomplished by performing the tasks in T_n in the order given.

Let S be a state, X be an axiom list, t be a compound task atom (which may or may not be ground), and m be the method:

(:method h [$name_1$] L_1 T_1 [$name_2$] L_2 T_2 ... [$name_n$] L_n T_n)

Suppose there is an mgu u that unifies t with h and suppose that m has a precondition L_i such that S and X satisfy L_i^u (If there is more than one such precondition, then let L_i be the first such precondition). Then we say that m is **applicable** to t in S and X, with the active precondition L_i and the active tail T_i. Then the result of applying m to t is the following set of task lists:

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

Each task list r in R is called a **simple reduction** of t by branch i of m in S and X. Figure 10 shows 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))
v	()
$(o^u)^v$	(:operator (!set-money john 40 35) ((has-money john 40)) ((has-money john 40)) ((has-money john 35)))
$(h^u)^v$	(!set-money john 40 35)
Result(S, $(h^u)^v$)	((has-money john 35) (has-money mary 30))
Result(S, $(o^u)^v$)	((has-money john 35) (has-money mary 30))

Figure 8: An example of an operator being applied.

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	()
v	()
$(o^u)^v$	(:operator (!clear-locations) () ((forall (?l) ((location ?l) (not (truck-at ?t ?l))) ((location ?l)))) ())
$(h^u)^v$	(!clear-locations)
Result(S, $(h^u)^v$)	((location l1) (truck-at truck1 l1))
Result(S, $(o^u)^v$)	((location l1) (truck-at truck1 l1))

Figure 9: An example of an operator with forall keyword.

S	((has-money john 40) (has-money mary 30))
X	()
t	(transfer-money john mary 5)
M	(:method (transfer-money ?p1 ?p2 ?amount) ((has-money ?p1 ?m1) (has-money ?p2 ?m2) (call >= ?m1 ?amount)) ((!set-money ?p1 ?m1 (call - ?m1 ?amount)) (!set-money ?p2 ?m2 (call + ?m2 ?amount))))
u	((?p1 → john) (?p2 → mary) (?amount → 5))
h^u	(transfer-money john mary 5)
L_1^u	((has-money john ?m1) (has-money mary ?m2) (call >= ?m1 5))
T_1^u	((!set-money john ?m1 (call - ?m1 5)) (!set-money mary ?m2 (call + ?m2 5)))
v	((?m1 → 40) (?m2 → 30))
$(L_1^u)^v$	((has-money john 40) (has-money mary 30) (call >= 40 30))
$(T_1^u)^v$	((!set-money john 40 35) (!set-money mary 30 35))

Figure 10: A sample method.

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 L_i and the active tail T_i.
  M' = the task list produced by replacing t in M with a simple reduction of
t by branch i of m in S.
endif

Figure 11: Task reduction

5.2.3 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 that 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 in Figure 11.

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:

- Both M and P are empty.
- $T = (t_1\ t_2\ \ldots\ t_k)$ 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).
- $T = (t_1\ t_2\ \ldots\ t_k)$ is a list of tasks in M that have no predecessor, where t_i is a task in T that 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).

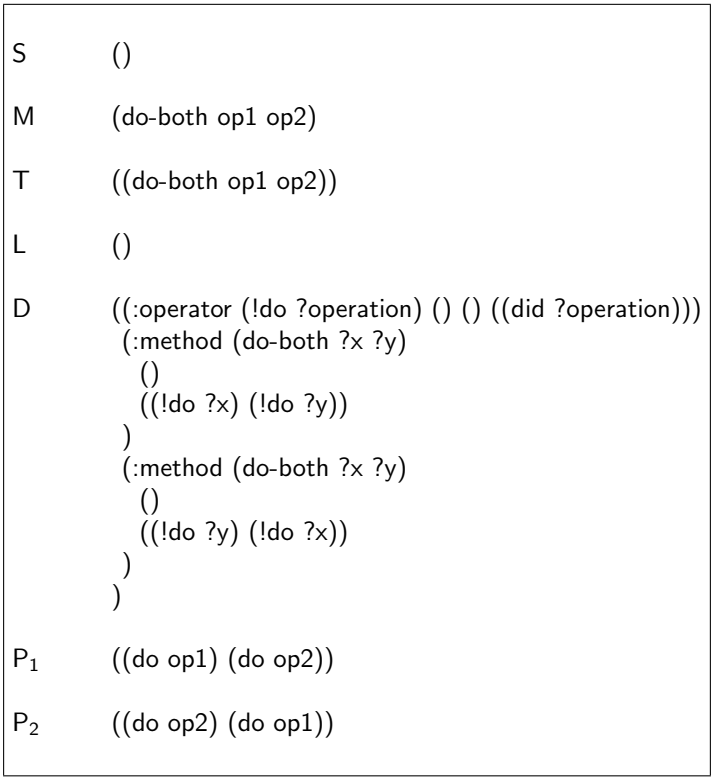

Figure 12: A sample planning problem.

– 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, in Figure 12, P_1 and P_2 are the only plans that solve (S, M, L, D).

Acknowledgments

This work was supported in part by the following grants and contracts: NSF IIS0412812 and Air Force Research Laboratory F30602-00-2-0505. The opinions expressed in this document are those of the authors and do not necessarily reflect the opinions of the funders.

We would also like to acknowledge the use of compiler generator software ANTLR (www.antlr.org) in JSHOP2 system.

A Implementing and Calling External Functions

External functions in JSHOP2 can be used in call terms. Several of these functions are built in the JSHOP2 code. However, users are free to implement their own functions and call them in their domains as long as they follow these steps:

- Create a class that implements Java interface Calculate (included in JSHOP2 code). The name of this class must be what user would like to use to invoke this function call from a domain. For example, a Java class named CheckNull will be used whenever a statement of the form (call CheckNull ...) is evaluated in a JSHOP2 domain.
- Implement the call function in the created class. This function has one parameter of the type List (which represents a list of terms), and returns a Term.
- Now the user can use call terms like (call CheckNull ...) in the domain. When evaluating this call term, JSHOP2 will invoke the call function in the CheckNull class. Note that it is user's responsibility to make sure that arguments in the call term are the ones that the Java function expects.

Note that the only places where JSHOP2 domains are case-sensitive are these user-defined code calls, since these identifiers are compiled directly to Java code (which is case-sensitive). For instance, in the above example, the letters of CheckNull must be of the same case in the code call.

Here is a list of built-in code calls in JSHOP2. They can be used directly in the domain description without any additional Java implementation: <, <=, =, >, >=, !=, +, -, *, /, ^, and Member.

For more information on the way these built-in functions are implemented and work, and also the Calculate interface, see the HTML documentation of JSHOP2 (doc/index.html) in the JSHOP2 distribution. To see examples of use of both built-in and user-defined code calls, see the examples in the examples directory of JSHOP2 distribution.

One important point to remember when implementing an external code call is that, for efficiency purposes, no matter how many call terms call a specific external code call, there will be only one object of the corresponding class representing all those call terms. Therefore, data members in such classes should be avoided, since they are shared by all the corresponding call terms in the domain.

B Implementing Comparator Functions to Be Used with Sorted Preconditions

A **comparator** function is a function that can be used to sort a list of symbols. There are two built-in comparator functions in JSHOP2, that can only be used with lists of numerical symbols: < and >. However, it is possible to implement user-defined comparator functions. In order to do so, the user needs to define a

class with the same name as the comparator function that implements the Java Comparator interface. This interface requires that a compare(Object o1, Object o2) function be implemented. In this case, the two objects o1 and o2 are of type Term[], which represents a binding as an array of terms. Moreover, this comparator class must have a constructor that accepts one parameter of type int. This is the index of the variable according to the value of which the sorting should take place (i.e., the index of the elements of o1 and o2 that should be used to compare those two bindings).

For more information on the way comparators are implemented and work, see the HTML documentation of JSHOP2 (doc/index.html) in the JSHOP2 distribution for classes CompLess and CompMore (the two classes that implement the two built-in comparator functions). To see examples of use of both built-in and user-defined comparators, see the examples in the examples directory of JSHOP2 distribution.

C Syntax Differences with SHOP2

Here is a list of syntax differences with SHOP2:

- In SHOP2, the operator precondition can be omitted. This is not the case with JSHOP2.
- The range of possible symbol names in JSHOP2 is not as large as in SHOP2 (See Subsection 3.1 for more details).
- Keywords :task, :ordered, and list are not supported anymore in JSHOP2 (They are all optional in SHOP2, so this does not decrease the expressive power of JSHOP2).
- eval terms are not allowed in JSHOP2. Also, all the structures that SHOP2 has kept for backward-compatibility only (such as make-problem and make-domain) are dropped in JSHOP2.
- There are several restrictions on where the call terms can not appear in SHOP2. There are no such restrictions in JSHOP2. Call terms may appear anywhere any other term may appear.
- The first satisfier precondition syntax in JSHOP2 is a little bit different from that of SHOP2. In JSHOP2, it is (:first L) where L is a logical expression, while in SHOP2 it is (:first L_1 L_2 ... L_n) where each L_i is a logical expression. In terms of expressive power, both forms are equal, but the former is more intuitive.
- Arbitrary LISP expressions can appear in two places in SHOP2: Assignment expressions and operator costs. These are replaced by arbitrary terms in JSHOP2. Also, in the sorted precondition syntax, the LISP expression is replaced with a user-defined or built-in comparison function.

- The syntax for the defproblem command allows for defining several planning problems in one expression in JSHOP2. This is not allowed in SHOP2.

References

[1] Okhtay Ilghami and Dana S. Nau. A general approach to synthesize problem-specific planners. Technical Report CS-TR-4597, Department of Computer Science, University of Maryland, October 2003.

[2] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–973. AAAI Press, 1999.

[3] Dana S. Nau, Héctor Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, WA, August 2001.