L2 Act: User Manual

L2Act: User Manual

Roni Khardon

TR-10-97

May 1997

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

L2Act: User Manual

Roni Khardon*
Aiken Computation Laboratory,
Harvard University,
Cambridge, MA 02138
roni@deas.harvard.edu

May 19, 1997

1 Introduction

This note describes the system L2Act, the options it includes, and how to use it. We assume knowledge of the general ideas behind the system [1], as well as some details on the implementation described in [2]. The system includes several on-line options, and several compile options. We start by describing these and then describe the structure of the input files, and how to use the system.

2 Soft Options

These are options that can be given on the command line. A typical invocation of the program looks like

```
learntoact input_file -MAXRUNS313 -LEVELWISE1 -SIGMA0.01
```

where `learntoact` is the name of the compiled program. All arguments apart from the input file start with a minus sign, then the option name follows in capital letters, and then the option value is a number. In the description of the option we give the range of values and (current) default value in parenthesis.

2.1 General Parameters

These options control whether the program will learn, test, and in what way:

TPRS (binary,0) When on, the program goes into test mode and expects a PRS in the input file instead of runs. In this mode, the second on-line parameter is the name of the file containing the runs to be tested upon.

TESTMODE (binary,0) In mode 0 the program will test whether the PRS solves the test problems. In addition, the average ratio of solution lengths produced by the PRS (when solved) to those that appear in the test file will be computed. In mode 1 the program will compute the percentage of "correct classification" on the actions that appear in the test runs, thus testing normal supervised learning classification accuracy. Mode 1 has not been in use for a while.

*Research supported by ONR grant N00014-95-1-0550, and ARO grant DAAL03-92-G-0115.

TLOOPS (binary, 0) In test mode 0, the program quits trying after some bound on the number of steps. Normally, when the PRS fails it is in an infinite loop doing and undoing the same thing. In TLOOPS mode the program tests whether an action has been repeated in the last 5 steps, and if so quits. This implementation is only partial !!! It is good for BW since an action should not be repeated, but not good enough for the logistics domain (although one can get fast and reasonable estimates with it).

TWIN (binary, 0) When on, the program goes to test mode, but expects a list of weights (and indices) as input. The indices and weights correspond to the rules in the standard enumeration. A weighted sum of votes is used to try to solve the runs. TESTMODE, and TLOOPS apply here as well.

WINNOW (integer, 0) Only values 3,4 are relevant and they indicate that the corresponding version of winnow should be run (in addition to the PRS algorithm). Currently, can only be used with standard enumeration.

LEVELWISE (binary, 0) When on, the levelwise algorithm is run instead of the standard one.

SIGMA (real, 0.01) When running the levelwise algorithm this parameter is used as the threshold.

2.2 Other General Parameters

DEBUG (integer,0) This flag is used to print various information useful for debugging. Use 17 to print the actions chosen in test mode, that is, the solutions produced by the PRS.

MAXRUNS (integer,0) The input files may contain many runs; only MAXRUNS is used in learning. When 0 all runs are used.

NODUPBIND (binary,0) When on, binding enumeration is restricted so that different variables are bound to different objects. (Note that this saves time for a given rule, but may require more rules.)

PREFMODE (integer,0) Several preference modes for choosing between rules are coded. Mode 0 and 2 are discussed in [2]. Others are variations.

NOBNDORDER (binary,0) Normally, the first binding that matches a rule is the one that determines which action it suggests. When on, this flag changes the semantics for learning. First, it is tested whether the rule can predict the correct action for some binding; if so it is considered correct. Otherwise, if it matches for any binding it is marked incorrect. This intuitively builds on the hope that all the bindings are good (and tries to avoid penalizing a rule just because of the binding order), but may reduce the number of negative examples for a rule.

PLAN (binary,1) When on, each rule includes all preconditions of the action. Mode 0 may not work with some options.

COUNTRULES (binary,1) Prints a count of rules in standard enumeration before starting.

CUTNWGT (integer, 0) This flag is used in Winnow to bound the number of weights that are printed as output (weights are sorted and larger is printed first). It is also used for debugging in various places; for normal operation it should be kept at 0.

2.3 Enumeration of Rules

Several parameters control the enumeration of rules; some refer to the standard enumeration (not using the levelwise algorithm).

KRULE (integer, 2) The width of the rules that are enumerated.

KSRULE (integer, 3) The width of rules for support predicates (must match the input, and all rules must be of same width).

KBIND (integer, 3) The maximal number of free variables in the rules. The same parameter is used for both support predicates and rules on the PRS.

KNRULE (integer, 2) In standard enumeration KNRULE is the number of positions which are allowed to use support predicates. The other positions only use the original predicates.

2.4 Filtering Rules

Several parameters control filtering of enumerated rules:

FILTER (binary, 1) General flag to control filtering. In PLAN mode it is tested that the body of the rule does not contradict any of the preconditions of the action.

NODUPVAR (binary, 1) filter a rule if it includes predicates that refer to the same variable, e.g. $on(1,1)$.

SFILT (binary, 1) In PLAN mode filter a rule if it includes a predicate that is a duplicate of a precondition of the action.

NOSKIPVAR (binary, 1) filter a rule if its variables are not a continuous set; e.g. filter rule if it only uses the variables 1,3 (since an equivalent rule that uses 1,2 is enumerated).

CONNECTED (binary, 1) filter a rule if its variables are not "connected". Connection between variables is introduced if they are mentioned in the same predicate. For example $on(1,2)clear(3)$ is not connected, but $on(1,2)on(2,3)clear(3)$ is.

FILTYPES (integer, 0) Use either 0 or 2. In this mode strong type checking is used, by identifying in the examples the types of the parameters of predicates, and filtering rules that contradict these type constraints. For example a rule with `AIRPORT(1)in(1,2)` will be filtered in the logistics domain since it is found that it never appeared with any object of type AIRPORT in the first parameter. The types are also used for binding enumeration. For each rule, the program computes which objects might bind to each variable and enumerates only these combinations.

In general type predicates (see below) will not be used in the rules, and this supplies a simple form of pruning the number of rules. When using FILTYPES there is a tension between using a predicate in the rules and using it as a type predicate. Due to implementation, when FILTYPES is used, type predicates cannot be used in the rules (even if hand coded). Namely, even if two types are not disjoint conjoining them is impossible. If the use of these predicates in the rules is desired one can produce copy of a predicate via the support rules (cf. `ap()` in the logistics domain).

3 Compile Options

Several options are controllable in compile time. To activate `#ifdef` option the compiler's command line should include `-Dparamname`. To set a value use `-Dparamname=number`.

- The possibility of using the levelwise algorithm is controlled by `#ifdef LWISE`.
- The standard enumeration algorithm uses a large table that saves the correct/cover results for each example-rule combination. This is controlled by `#ifdef RTAB`. When using the table, enough space must be assigned; the size is controlled by `#define NORULES 3400`, and `#define NOEXAMPLES 5000`.
- In winnow mode, number 4, more information must be remembered (which action was taken for each rule). This is enabled by `#ifdef WINBIG`.
- For the purpose of drawing learning curves, it is useful to compute an output for the learning algorithm after every so many examples. This is enabled by `#ifdef STEPEX`, and the particular numbers by `#define STEPBASE 200`, `#define STEPMAX 1200`, `#define STEPINC 200`.
- To compare all preference modes in a single run use `#ifdef ALLPREF`, and to control the number of preference modes that are compared use `#define NOPREF 6`
- For winnow we allow two parameters. First the number of versions of promotion parameters to use; by using `#define NOVERS 2` we get 3 versions starting with 0.5 and equally spaced to 1 (not including 1). We also allow to repeat winnow on the same set of examples. This is done by `#define ITER 10`.

3.1 Compilation Examples

The code is contained in 5 files: `l2a.c rules.c learn.c parseinp.c mydefs.h`. The file `l2a.c` handles the command line options, `rules.c` includes the code and learning algorithm for the levelwise mode, `learn.c` contain the standard learning algorithm as well as many routines, and `parseinp.c` contains the parser as well as the code of the test programs.

- To run with 1200 examples in 4 steps (for learning curve) using the levelwise algorithm with an enumeration that uses no more than 11000 rules use:

  ```
  gcc -DSTEPEX -DSTEPBASE=300 -DSTEPMAX=1200 -DSTEPINC=300 -DLWISE -DRTAB
  -DNORULES=11000 -DNOEXAMPLES=1230 l2a.c rules.c learn.c parseinp.c
  -O3 -o learntoact
  ```

  When running, one has to use a number of runs that supply at least `STEPMAX` steps and at most `NOEXAMPLES` steps.
- The above option is the general one and it allows to run the standard algorithm as well as Winnow3. If the levelwise algorithm is not used then (to decrease the size of the code) just omit `-DLWISE` and `rules.c` in the above.
- To use Winnow4 with 10 iterations, and 2 versions use:

  ```
  gcc -DITER=10 -DNOVERS=2 -DWINBIG -DRTAB -DNORULES=9000 -DNOEXAMPLES=370
  l2a.c learn.c parseinp.c -O3 -o learntoact
  ```

- For testing we do not need the big table or `rules.c`; use:

  ```
  gcc l2a.c learn.c parseinp.c -O3 -o testl2a
  ```

4 The Parser and the Input File

Examples for input files for learning and testing appear in the appendix. Here are some notes regarding the parsing:

- The parser is somewhat fragile, basing its decisions not on the parenthesis in the inputs but rather on the line structure (which is sufficient).
- Everything up to the first line starting with a number sign is ignored by the parser.
- The file includes several sections to be parsed; each section is marked by a number sign.
- The first section `#predicates` includes predicates' names and parameters (the names of parameters are irrelevant here, but are needed to determine the arity). Note: *each item is expected on a separate line.*
- The next section `#typepreds` is optional, and has the same structure as `#predicates`. In normal mode these predicates will simply not be used in the rules. In FILTYPES mode they are also used for type checking (it is tested for other predicates which type predicates are accepted as parameters).
- The section `#operators` describes the actions in the domain, and is based on the `.lisp` style in GraphPlan. Note that (as in GraphPlan) only positive literals are expected as part of the operators. There are two important differences though:

  1. The first part of the preconditions of actions, includes "object generation predicates" for GraphPlan. In the original format they look like `(<ob1> OBJECT)` but we expect them in reverse order i.e. `(OBJECT <ob1>)`.
  2. The parser ignores all parenthesis etc. and parses the operators based on the line structure, and the general structure. Namely, it is expected that the operator will have the following structure:

```
(OPERATOR
  OPNAME
  (params  LIST OF PARAM NAMES )
   **** only one line expected ***
  (preconds
    LIST OF (PREDICATE PRED-OBJECTS)
      **** each on a separate line ****
    the "and" construct is optional (currently ignored).
  (effects
     ONE LINE IS IGNORED AFTER THE KEYWORD effects
     then LIST OF EFFECTS
    del/add (PREDICATE PRED-OBJECTS)
     **** each on a separate line ****
    is expected. All the del effects must appear before
    the add effects.
```

- The section `#supportpreds` includes pairs of rules that generate new predicates. The first rule is evaluated once on all bindings, and the second is reevaluated until no more changes occur (as in [1]). Note that *each rule is expected on a separate line and a line of space between pairs is expected.*

- Then, any number of descriptions of runs appear. Each such description includes several sections.

- The section `#RUN` is empty; just used to signal the boundary.

- The section `#objects` includes a list of object names that appear in this run. As before *each object is expected on a separate line* .

- The section `#start` includes a list of literals that are satisfied in the beginning of the run. Note that *all other predicates are assumed to be 0*, namely, we are using the so-called closed world assumption in the start state. Here again *each literal is expected on a separate line* .

- The section `#goal` includes a list of literals that should be satisfied in the end of the run. Note that (as in GraphPlan) only positive goal literals are considered. Here, however, we are not using the closed world assumption and the rest of the predicates may take any value in the goal. As before *each literal is expected on a separate line* .

- The section `#actions` includes a list of actions in the form:
  `DUMMYWORD OPNAME OBJNAME OBJNAME ...`
  that achieve the goal. Again *each action is expected on a separate line* .

- In test mode, instead of runs, the program expects a PRS or weights from winnow. These are announced by `#TESTS`. In Normal mode the PRS just follows.

- In `FILTYPES` mode the program expects information on legal types before the PRS and another marker `#TESTS` in between. (These are automatically generated so the details are omitted).

- In `TWIN` mode, the program expects the names of the type predicates used to be printed before the weights, and again another marker `#TESTS` is expected. (These are automatically generated so the details are omitted).

5 Utility Programs

Several utility programs to prepare examples, run learning experiments and tests, as well as for tabulating the results have been written. I will only mention one; the program `mktfiles.c` takes the output of the learning algorithm that may include several PRS from various options, and creates from it test files as needed. For learning experiments where `FILTYPES` was used the program `mktfiles-legal.c` includes the additional required information.

6 How to use

Here is one example of learning and testing

```
learntoact log_file.runs -MAXRUNS114 -LEVELWISE1 -KRULE3 -KBIND5
-FILTYPES2 > output_file

mktfiles-legal output_file
```

This creates several files called `Toutput_file.p0`, `Toutput_file.p1`, etc.

```
testl2a Toutput_file.p0 test_runs_file -TPRS1 -TLOOPS0 -FILTYPES2
-KRULE3 -KBIND5
```

A Input Files

A.1 Blocks World

```
#predicates

 (arm-empty)
 (on-table <ob1>)
 (clear <ob1>)
 (holding <ob1>)
 (on <ob> <underob>)

#typepreds

 (OBJECT <ob1>)

#operators

(OPERATOR
  PICK-UP
  (params <ob1>)
  (preconds
   ((OBJECT <ob1>))
   (and (clear <ob1>)
        (on-table <ob1>)
        (arm-empty)))
  (effects
      () ; no vars need genenerated in effects list
      ((del (on-table <ob1>))
       (del (clear <ob1>))
       (del (arm-empty))
       (add (holding <ob1>)))))

(OPERATOR
  PUT-DOWN
  (params <ob>)
  (preconds
```

```
   ((OBJECT <ob>))
   (holding <ob>))
  (effects
        ()
((del (holding <ob>))
 (add (clear <ob>))
 (add (arm-empty))
 (add (on-table <ob>))))))


(OPERATOR
 STACK
 (params <ob> <underob>)
 (preconds
   ((OBJECT <ob>)
    (OBJECT <underob> ))
    (and (clear <underob>)
 (holding <ob>)))
 (effects
      ()
      ((del (holding <ob>))
       (del (clear <underob>))
       (add (arm-empty))
       (add (clear <ob> ))
       (add (on <ob> <underob>))))))

(OPERATOR
 UNSTACK
 (params <ob> <underob>)
 (preconds
  ((OBJECT <ob> )
   (OBJECT <underob> ))
  (and (on <ob> <underob>)
       (clear <ob>)
       (arm-empty)))
 (effects
       ()
((del (on <ob> <underob>))
 (del (clear <ob>))
 (del (arm-empty))
 (add (holding <ob>))
 (add (clear <underob>))))))



#supportpreds
```

```
Base Rule: G(on(1,2)) G(on(1,2)) G(on(1,2)) ==> ingoal(1)
Recursive Rule: G(on-table(1)) G(on-table(1)) G(on-table(1)) ==> ingoal(1)

Base Rule: G(on-table(1)) on-table(1) on-table(1) ==> inplacea(1)
Recursive Rule: inplacea(2) G(on(1,2)) on(1,2) ==> inplacea(1)

Base Rule: G(on(1,2)) on(1,2) ^_ingoal(2) ==> inplaceb(1)
Recursive Rule: inplaceb(2) G(on(1,2)) on(1,2) ==> inplaceb(1)

Base Rule: on(1,2) on(1,2) on(1,2) ==> above(1,2)
Recursive Rule: above(2,3) on(1,2) on(1,2) ==> above(1,3)




#RUN

#objects

1
2
3
4
5
6
7
8

#start

(preconds
(OBJECT 1)
(OBJECT 2)
(OBJECT 3)
(OBJECT 4)
(OBJECT 5)
(OBJECT 6)
(OBJECT 7)
(OBJECT 8)
(on 1 5)
(on-table 2)
(clear 2)
(on 3 7)
(on 4 6)
(on 5 3)
(on 6 1)
(on-table 7)
```

```
(on 8 4)
(clear 8)
(arm-empty))



#goal

(effects
(on 1 4)
(on 2 3)
(on 4 7)
(on 5 2)
(on 7 5)
(on-table 8)
)


#actions

1 UNSTACK_8_4
2 PUT-DOWN_8
3 UNSTACK_4_6
4 PUT-DOWN_4
5 UNSTACK_6_1
6 PUT-DOWN_6
7 UNSTACK_1_5
8 PUT-DOWN_1
9 UNSTACK_5_3
10 PUT-DOWN_5
11 UNSTACK_3_7
12 PUT-DOWN_3
13 PICK-UP_2
14 STACK_2_3
15 PICK-UP_5
16 STACK_5_2
17 PICK-UP_7
18 STACK_7_5
19 PICK-UP_4
20 STACK_4_7
21 PICK-UP_1
22 STACK_1_4

#RUN

...
```

A.2 The Logistics Domain

```
#predicates

 (at <truck> <loc>)
 (in <obj> <truck>)
 (loc-at <loc-from> <city>)

#typepreds

 (OBJECT <obj>)
 (TRUCK <truck>)
 (LOCATION <loc>)
 (AIRPLANE <airplane>)
 (AIRPORT <loc-from>)
 (CITY <city>)

#operators

(OPERATOR
  LOAD-TRUCK
  (params <obj> <truck> <loc>)
  (preconds
   ((OBJECT <obj> )
   (TRUCK <truck> )
   (LOCATION <loc> ))
   (and (at <truck> <loc>)
(at <obj> <loc>)))
  (effects
   ()
   ((del (at <obj> <loc>))
    (add (in <obj> <truck>)))))

(OPERATOR
  LOAD-AIRPLANE
  (params <obj> <airplane> <loc>)
  (preconds
   ((OBJECT <obj>)
   (AIRPLANE <airplane>)
   (LOCATION <loc>))
   (and (at <obj> <loc>)
(at <airplane> <loc>)))
  (effects
   ()
```

```
   ((del (at <obj> <loc>))
    (add (in <obj> <airplane>)))))

(OPERATOR
  UNLOAD-TRUCK
  (params <obj> <truck> <loc>)
  (preconds
   ((OBJECT <obj>)
   (TRUCK <truck>)
   (LOCATION <loc>))
   (and (at <truck> <loc>)
(in <obj> <truck>)))
  (effects
   ()
   ((del (in <obj> <truck>))
    (add (at <obj> <loc>)))))

(OPERATOR
  UNLOAD-AIRPLANE
  (params <obj> <airplane> <loc>)
  (preconds
   ((OBJECT <obj>)
   (AIRPLANE <airplane>)
   (LOCATION <loc>))
   (and (in <obj> <airplane>)
(at <airplane> <loc>)))
  (effects
   ()
   ((del (in <obj> <airplane>))
    (add (at <obj> <loc>)))))

(OPERATOR
  DRIVE-TRUCK
  (params <truck> <loc-from> <loc-to> <city>)
  (preconds
   ((TRUCK <truck>)
   (LOCATION <loc-from>)
   (LOCATION <loc-to>)
   (CITY <city> ))
   (and (at <truck> <loc-from>)
        (loc-at <loc-from> <city>)
        (loc-at <loc-to> <city>)))
  (effects
   ()
   ((del (at <truck> <loc-from>))
    (add (at <truck> <loc-to>)))))
```

```
(OPERATOR
  FLY-AIRPLANE
  (params <airplane> <loc-from> <loc-to>)
  (preconds
   ((AIRPLANE <airplane>)
   (AIRPORT <loc-from>)
   (AIRPORT <loc-to>))
   (at <airplane> <loc-from>))
  (effects
   ()
   ((del (at <airplane> <loc-from>))
    (add (at <airplane> <loc-to>)))))

#supportpreds


Base Rule: AIRPORT(1) AIRPORT(1) AIRPORT(1) ==> ap(1)
Recursive Rule: AIRPORT(1) AIRPORT(1) AIRPORT(1) ==> ap(1)



#RUN

#objects

package1
package2
bos-truck
pgh-truck
la-truck
airplane1
airplane2
bos-po
pgh-po
la-po
bos-airport
pgh-airport
la-airport
bos
pgh
la


#start


(preconds
```

```
(OBJECT package1 )
(OBJECT package2 )
(TRUCK bos-truck )
(TRUCK pgh-truck )
(TRUCK la-truck )
(AIRPLANE airplane1 )
(AIRPLANE airplane2 )
(LOCATION bos-po )
(LOCATION pgh-po )
(LOCATION la-po )
(AIRPORT bos-airport )
(LOCATION bos-airport )
(AIRPORT pgh-airport )
(LOCATION pgh-airport )
(AIRPORT la-airport )
(LOCATION la-airport )
(CITY bos )
(CITY pgh )
(CITY la )
 (loc-at pgh-po pgh)
 (loc-at pgh-airport pgh)
 (loc-at bos-po bos)
 (loc-at bos-airport bos)
 (loc-at la-po la)
 (loc-at la-airport la)
(at package1 pgh-po)
(at package2 pgh-po)
(at airplane1 pgh-airport)
(at airplane2 pgh-airport)
(at bos-truck bos-airport)
(at pgh-truck pgh-po)
(at la-truck la-po)
 )



#goal


(effects
(at package1 la-po)
(at package2 la-po)
 )


#actions
```

```
1 LOAD-TRUCK package1 pgh-truck pgh-po
2 LOAD-TRUCK package2 pgh-truck pgh-po
3 DRIVE-TRUCK pgh-truck pgh-po pgh-airport pgh
4 UNLOAD-TRUCK package1 pgh-truck pgh-airport
5 LOAD-AIRPLANE package1 airplane1 pgh-airport
6 UNLOAD-TRUCK package2 pgh-truck pgh-airport
7 LOAD-AIRPLANE package2 airplane1 pgh-airport
8 FLY-AIRPLANE airplane1 pgh-airport la-airport
9 UNLOAD-AIRPLANE package1 airplane1 la-airport
10 UNLOAD-AIRPLANE package2 airplane1 la-airport
11 DRIVE-TRUCK la-truck la-po la-airport la
12 LOAD-TRUCK package1 la-truck la-airport
13 LOAD-TRUCK package2 la-truck la-airport
14 DRIVE-TRUCK la-truck la-airport la-po la
15 UNLOAD-TRUCK package1 la-truck la-po
16 UNLOAD-TRUCK package2 la-truck la-po

#RUN

...
```

A.3 Input File in Test Mode

The file should include everything up to `#RUN` in the input file. Next follows the PRS. Here is how it looks for the logistics domain (including the type predicates information). The file including test runs (the second parameter in test mode) should include runs using the same structure as above. (Domain description is not necessary, since it is read from the first input file, but it can be included since everything until the first `#RUN` is ignored.)

```
#TESTS

0 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
1 0 0 0
0 1 0 0
1 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
```

```
0 1 0 0
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
0 0 0 0
1 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 0 0 0
0 0 0 0
1 0 0 0
0 0 0 0
0 0 0 0
```

#TESTS

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) in(1,2)
G(at(1,3)) G(at(1,3)) G(at(1,3)) ==> UNLOAD-TRUCK(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) in(1,2) at(2,3)

```
G(at(1,4)) loc-at(3,5) loc-at(4,5) ==> UNLOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) in(1,2) at(2,3)
G(at(1,3)) G(at(1,3)) G(at(1,3)) ==> UNLOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) AIRPLANE(2) LOCATION(3) at(1,3) at(2,3)
G(at(1,4)) ^_loc-at(3,5) loc-at(4,5) ==> LOAD-AIRPLANE(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) at(1,3)
G(at(1,4)) loc-at(3,5) loc-at(4,5) ==> LOAD-TRUCK(1,2,3)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) at(1,3)
G(at(1,4)) loc-at(4,5) ^_ap(3) ==> LOAD-TRUCK(1,2,3)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4)
G(at(5,3)) in(5,1) in(5,1) ==> DRIVE-TRUCK(1,2,3,4)

Rule: OBJECT(1) TRUCK(2) LOCATION(3) at(2,3) in(1,2)
G(at(1,4)) loc-at(4,5) ap(3) ==> UNLOAD-TRUCK(1,2,3)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4)
in(5,1) in(5,1) in(5,1) ==> DRIVE-TRUCK(1,2,3,4)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
G(at(5,3)) ^_in(4,1) in(5,1) ==> FLY-AIRPLANE(1,2,3)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4)
G(at(5,2)) at(5,3) at(5,3) ==> DRIVE-TRUCK(1,2,3,4)

Rule: TRUCK(1) LOCATION(2) LOCATION(3) CITY(4) at(1,2) loc-at(2,4) loc-at(3,4)
at(5,3) ap(2) ap(2) ==> DRIVE-TRUCK(1,2,3,4)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
at(4,3) G(at(4,5)) ^_ap(5) ==> FLY-AIRPLANE(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
at(4,3) G(at(4,5)) G(at(4,5)) ==> FLY-AIRPLANE(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
G(at(4,2)) at(4,3) at(4,3) ==> FLY-AIRPLANE(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
^_at(5,3) G(at(5,3)) ^_in(4,5) ==> FLY-AIRPLANE(1,2,3)

Rule: AIRPLANE(1) AIRPORT(2) AIRPORT(3) at(1,2)
^_at(4,3) in(5,1) in(5,1) ==> FLY-AIRPLANE(1,2,3)
```

References

[1] R. Khardon. Learning to take actions. In *Proceedings of the National Conference on Artificial Intelligence*, pages 787–792, 1996.

[2] R. Khardon. Learning action strategies for planning domains. Technical Report TR-09-97, Aiken Computation Lab., Harvard University, May 1997.