# Applying Automatic Planning Techniques to Airport Ground-Traffic Control — A Feasibility Study

Sebastian Trüg, Jörg Hoffmann, and Bernhard Nebel

Institut für Informatik
Universität Freiburg
79110 Freiburg, Germany
⟨*last name*⟩@informatik.uni-freiburg.de

**Abstract.** Planning techniques have been matured as demonstrated by the performance of automatic planning systems at recent international planning system competitions. Nowadays it seems feasible to apply planning systems to real-world problems. In order to get an idea what the performance difference between special-purpose techniques and automatic planning techniques is, we applied these techniques to the airport traffic control problem and compared it with a special purpose tool. In addition to a performance assessment, this exercise also resulted in a domain model of the airport traffic control domain, which is currently used as a benchmark in the 4th International Planning Competition.

**Technical Report Nr. 199**

**Report Date April 15, 2004**

## 1 Introduction

Planning techniques have matured as demonstrated by the performance of automatic planning systems at recent international planning system competitions [1, 12]. Current planning systems are able to generate plans with hundred steps and more compared with ten steps or less ten years ago. The reason for this performance boost is partly the increase in computing power and mainly the use of new algorithms and the development of powerful heuristics [2, 3].

In order to assess how feasible it is to apply automatic planning systems to real-world problems, we used these techniques to solve the operational airport ground-traffic control problem [8]. This problem can be characterized as a job-shop scheduling problem with blocking. Using special-purpose techniques, it can be solved approximately for a realistic number of airplanes (roughly 50) on realistic airports (such as the Frankfurt airport).

So, how far do we get with planning techniques for this problem? As our experiments indicate, the *expressiveness* of current planning systems approaches the point where one can formulate the traffic problem as a planning problem. From a *performance* point of view, the results are less encouraging. The current state of the art is only good enough for small airports with few airplanes. One has to put this result into perspective, however. Special-purpose scheduling techniques are, of course, highly optimized for the particular scheduling problem. Furthermore, the performance penalty to be expected for general-purpose solutions is always quite high and often, as in our case, prohibitive. On the positive side, the exercise of formalizing the traffic problem has led to a new challenging, real-world benchmark problem for automatic planning systems. Indeed, the domain is used as a benchmark in IPC-4, the 4th International Planning Competition.

The rest of the paper is structured as follows. In Section 2, we give a description of the airport ground-traffic control problem. Section 3 describes then the formalization using PDDL [11], the *de facto* standard for planning systems. In order to allow existing planning systems to be used, we also describe how to compile the PDDL specification into basic STRIPS. Based on that, in Section 4, the results of the experiments are given. Section 5 summarizes and concludes. An appendix contains the full PDDL descriptions of (the ADL formulations of) the non-temporal version, and the temporal version, of our domain.

## 2 The Airport Ground-Traffic Control Problem

In a nutshell, the airport ground-traffic control problem consists of coordinating the movements of airplanes on the airport so that they reach their planned destinations (runway or parking position) as fast as possible – whereby collisions shall be, of course, avoided.

The airplanes move on the airport *infrastructure*, which consists of *runways, taxi ways*, and *parking positions*. Airplanes are generally divided into the three `Wake Vortex Categories' : *light, medium*, and *heavy*, which classify them according to their engine exhaust. A moving airplane can either be *in-bound* or *out-bound*. In-bound airplanes are recently landed and are on their way from the runway to a parking position, usually a gate. Out-bound airplanes are ready for departure, meaning they are on their way to the departure runway. Since airplanes are not able to move backwards, they need to be pushed back from the gate on the taxiway where they start up their engines. Some airports provide different park positions that allow an airplane to start its engines directly, but to simplify the situation, we assume that an airplane always needs to be pushed back.

The ground controller has to communicate to the airplanes which ways they shall take and when to stop. While such guidance can be given purely reactively, it pays off to base decisions on anticipating the future. Otherwise it may happen that airplanes block each other and need more time than necessary to reach their destinations on the airport. The objective is to minimize the overall summed up traveling times of all airplanes.

From a formal point of view, one considers the problem with a time horizon of say one hour and schedules all movements minimizing the movement times of the planes. Of course, because the situation changes continually (new planes arrive and schedules cannot be executed as planned), continuous rescheduling is necessary. We will consider, however, only the static optimization problem with a given situation on the airport and a time horizon of a fixed time span.

Our domain representation and implementation is based on software by Wolfgang Hatzack, namely on a system called *Astras*: Airport Surface ground TRAffic Simulator. This is a software package that was originally designed to be a training platform for airport controllers. Astras provides a two-dimensional view of the airport, allowing the user to control the airplanes by means of point and click. Astras also includes features for simulating the traffic flow on an airport over the course of a specified time window, as well as an automated controller (named *Acore*) driven by a greedy re-scheduling approach [8]. Our PDDL domain encoding is based on Astras's internal representation of airports. We generated our test instances by software that is integrated with Astras. During an airport simulation, if desired by the user, our software exports the current traffic situation in various PDDL encodings.

## 3 The PDDL Encoding of the Airport Domain

The central object in the PDDL encoding of the airport domain is the airplane that moves over the airport infrastructure. The airport infrastructure is built out of segments. An airplane always occupies one segment and may block several others depending on its type. Our assumption here is that medium and heavy airplanes block the segment behind them whereas light airplanes only block the segment they occupy. Blocked segments cannot be occupied by another airplane. To handle terms like *behind* we need to introduce direction in segments. Since our segments are two-dimensional objects we need exactly two directions which we quite inappropriately call north and south. Every segment has its north end and its south end so it becomes possible to talk about direction in a segment.

To model the airplane movement we need at least two actions. The *move* action describes the normal forward movement of an airplane from one segment to another. The *pushback* action describes backward movement when an airplane is being pushed back from its park position.

We also introduce an airplane state. In our simplified version of an airport we have three states. An airplane can either be moving, be pushed, or be parked. The reason for these states is simple. We want to make sure that an airplane only moves backwards while being pushed from its park position and only moves forward if not. The parked state is necessary since a parked airplane's engines are off and thus the airplane does not block any segments except the one it occupies unlike when moving.

The actions *park* and *startup* describe the transitions between the different states. As one may expect the park action makes sure the airplane only blocks the occupied segment while the startup action does the exact opposite. It initially blocks segments depending on the airplane type.

A last action is needed to completely remove the airplane from the airport after takeoff. This action is called *takeoff* and makes sure the airplane does not block or occupy any segment anymore.

In the following we describe our different encodings of the airport domain: a durative and non-durative ADL [14] encoding, a STRIPS [6] encoding where the ADL constructs were compiled away [13], finally a means to model runway blocking for landing airplanes.

### 3.1 ADL Encoding

Our domain has four types of objects: *airplane*, *segment*, *direction*, and *airplanetype*.

The airplane type (its Wake Vortex Category) is described with the *has-type* predicate:

```
(has-type ?a - airplane ?t - airplanetype)
```

The airplane state is described with four predicates:

```
(airborne ?a - airplane ?s - segment)
(is-moving ?a - airplane)
(is-pushed ?a - airplane)
(is-parked ?a - airplane ?s - segment)
```

The *is-parked* predicate has a second parameter stating the park position the airplane is parked at. The second parameter of the *airborne* predicate just states from which segment the airplane took off. This is useful in case an airport provides several departure runways and we want to force an airplane to use a specific one. Apart from the airplane state we also need to describe the current position of an airplane and its heading:

```
(at-segment ?a - airplane ?s - segment)
(facing ?a - airplane ?d - direction)
```

We need several predicates to describe the airport structure. All the following predicates are static; they will be set once in the initial state and never be changed.

```
(can-move ?s1 ?s2 - segment ?d - direction)
(can-pushback ?s1 ?s2 - segment ?d - direction)
(move-dir ?s1 ?s2 - segment ?d - direction)
(move-back-dir ?s1 ?s2 - segment ?d - direction)
(is-blocked ?s1 - segment ?t - airplanetype ?s2 - segment ?d - direction)
(is-start-runway ?s - segment ?d - direction)
```

The *can-move* predicate states that an airplane may move from segment *s1* to segment *s2* if its facing direction *d*. The *can-pushback* predicate describes the possible backward movement which is similar to the *can-move* predicate. In our encodings, the *can-move* predicate holds only for pairs of segments that belong to the *standard routes* on the airport – this is common practice in reality (as reroutes are likely to cause trouble or at least confusion), and yields much better planner performance.

The *move-dir* and *move-back-dir* predicates state the airplane's heading after moving from segment *s1* to segment *s2*. That means that in a correct airport domain fact file every *can-move* (*can-pushback*) predicate has its *move-dir* (*move-back-dir*) counterpart.

The *is-blocked* predicate is used to handle all blocking. It says that segment *s1* will be blocked by an airplane of type *t* at segment *s2* facing into direction *d*. We will see the use of this predicate in the move action's effect in detail.

The last predicate, *is-start-runway*, states that an airplane at segment *s* facing direction *d* is allowed to takeoff. This is an essential predicate as we cannot allow an airplane to takeoff wherever it (or better: the planner) wants.

Finally, we need two predicates to describe the current situation on the airport regarding blocked and occupied segments:

```
(occupied ?s - segment)
(blocked ?s - segment ?a - airplane)
```

```
(:action move
 :parameters (?a  - airplane
              ?t  - airplanetype
              ?d1 - direction
              ?s1 - segment
              ?s2 - segment
              ?d2 - direction)
 :precondition (and
                  (has-type ?a ?t)
                  (is-moving ?a)
                  (at-segment ?a ?s1)
                  (facing ?a ?d1)
                  (can-move ?s1 ?s2 ?d1)
                  (move-dir ?s1 ?s2 ?d2)
                  (not (exists (?a1 - airplane) (and (not (= ?a1 ?a))
                                                     (blocked ?s2 ?a1))))
                  (forall (?s - segment) (imply (and (is-blocked ?s ?t ?s2 ?d2)
                                                     (not (= ?s ?s1)))
                                                (not (occupied ?s))
                  ))
               )
 :effect       (and
                  (at-segment ?a ?s2)
                  (not (at-segment ?a ?s1))
                  (occupied ?s2)
                  (not (occupied ?s1))
                  (when (not (= ?d1 ?d2))
                        (facing ?a ?d2))
                  (when (not (= ?d1 ?d2))
                  (blocked ?s2 ?a)
                  (not (facing ?a ?d1)))
                  (when (not (is-blocked ?s1 ?t ?s2 ?d2))
                        (not (blocked ?s1 ?a)))
                  (forall (?s - segment) (when (is-blocked ?s ?t ?s2 ?d2)
                                               (blocked ?s ?a)
                  ))
                  (forall (?s - segment) (when (and (is-blocked ?s ?t ?s1 ?d1)
                                                    (not (= ?s ?s2))
                                                    (not (is-blocked ?s ?t ?s2 ?d2))
                                               )
                                         (not (blocked ?s ?a))
                  ))
               )
)
```

**Fig. 1.** ADL *move* action

**3.1.1 Non-durative actions** Now we will look at the actions used in the non-durative ADL domain in detail.

The most important one is clearly the *move* action (see Figure 1). The parameters have the following meaning: an airplane *a* of type *t* facing direction *d1* on segment *s1* moves to segment *s2* and then faces direction *d2*. The first four predicates in the precondition make sure the airplane is in a proper state, meaning it is located in the right segment facing the right direction. Predicates five and six represent the airport structure as described above. The last two formulas are the really interesting ones as they deal with segment blocking. The first one makes sure that no other airplane blocks the segment our airplane is moving to. Here we use the full power of ADL to check for every airplane if it is blocking segment *s2*. Our moving airplane cannot block itself so

```
(:action startup
  :parameters (?a - airplane ?t - airplanetype ?s - segment ?d - direction)
  :precondition (and
                 (is-pushing ?a)
                 (has-type ?a ?t)
                 (at-segment ?a ?s)
                 (facing ?a ?d)
                 (forall (?s1 - segment)
                         (imply (and (is-blocked ?s1 ?t ?s ?d)
                                     (not (= ?s ?s1)))
                                (not (occupied ?s1))
                 ))
                )
  :effect       (and
                 (not (is-pushing ?a))
                 (is-moving ?a)
                 (forall (?s1 - segment)
                         (when (is-blocked ?s1 ?t ?s ?d)
                               (blocked ?s1 ?a)
                 ))
                )
)
```

**Fig. 2.** ADL *startup* action

we exclude it from the check. The second formula is used to make sure that none of the segments our airplane will block after the movement is occupied by another airplane. To achieve that we use the *is-blocked* predicate. Its instances give us the segments that will be blocked once our airplane has moved to segment *s2*. If we find such a segment we make sure its not occupied since a segment may not be occupied and blocked by different airplanes at the same time. We skip segment *s1* since it is always occupied by our airplane.

If all of these conditions apply, the *move* action may be executed. Most of the effects should be self-explanatory: updating of the occupied segment, changing the heading if necessary, and blocking the occupied segment. Again, the blocking is the part where full ADL is needed. We iterate over all segments to find those that are blocked from our airplane after the movement and those that were blocked before the movement. The latter ones need to be unblocked with exception of those blocked after the movement. Again, the *is-blocked* predicate is the central tool.

The *pushback* action differs only little from the *move* action. That is why we will skip a detailed description here and take a look at one of the more interesting actions used for performing the transitions of the airplane state.

The *startup* action (see Figure 2) represents the process of starting the airplane's engines after it has been pushed back from its park position and thus is clearly only needed for out-bound airplanes. That is why the preconditions contain the *is-pushing* predicate. The *startup* action represents the process of the airplane starting its engines. That means it begins blocking segments. So we need the exact same check for occupied segments as in the *move* action and also the blocking formula in the action's effect. Apart from that we only update the airplane state to *is-moving*.

The *park* action does the opposite of the *startup* action by unblocking all segments except the occupied one.

The takeoff action makes sure the airplane is completely removed from the airport meaning it does not block or occupy any segments anymore.

We now define an invariant for an airplane to prove the correctness of the defined actions. The airplane type is irrelevant in this context since it is not changed by any of the actions so we leave it out to simplify the invariant.

**Definition 1.** *A correct state of an airplane $a$ is defined by the following facts:*

1. *$a$ is at exactly one segment or is airborne.*
2. *$a$ occupies exactly the segment it is at or is airborne.*
3. *If $a$ is airborne it neither occupies nor blocks any segments.*
4. *$a$ is facing in exactly one direction or is airborne.*
5. *If $a$ is moving or being pushed it only blocks the segments determined by the is-blocked predicate and the one it occupies.*
6. *If $a$ is parked it only blocks the segment it occupies.*
7. *$a$ never blocks a segment occupied by another airplane.*
8. *$a$ never occupies a segment blocked by another airplane.*

**Proposition 1.** *If all airplanes had a correct state before executing an action of the airport domain they also have correct states afterwards.*

**3.1.2 Durative actions** Now we will enhance the domain by means of durative actions. The main thing to do is determine the duration of every action. The duration of both the *move* and the *pushback* action is calculated from the length of the segment the airplane is moving from and a fixed velocity. Since the startup action represents the process of an airplane starting its engines, the duration of this action is based on the number of engines the airplane has. The *park* and the *takeoff* action's durations are both set to fixed values.

To represent the length of a segment and the number of engines an airplane has we need two functions:

```
(length ?s - segment)
(engines ?a - airplane)
```

The correctness of the durative actions can be proved the same way as the non-durative versions since all blocking and occupations conditions are required to be true during the full length of the action.

### 3.2 STRIPS

Most current PDDL planning systems can not handle full ADL, especially durative ADL. To work around this problem and make the airport domain accessible to most planning systems, we need to compile away the ADL constructs to create a STRIPS only version. We will do this by pre-instantiating the more complicated actions to remove any conditional effects and quantified variables.

Due to the lack of quantified variables there is no way to determine which segments need to be unblocked and unoccupied when moving. The solution to this problem is pre-instantiating. It means that we create one *move* (and *pushback*) action for every pair of segments. This way we can explicitly test the involved segments without using the *is-blocked* predicate. But this is not enough. Since blocking is based on the airplane type and the current heading of the airplane both these variables need to be pre-instantiated, too. Finally, even the resulting heading needs to be pre-instantiated because we would need conditional effects which STRIPS does not offer. This leaves us with a single parameter for the *move* and *pushback* actions: the airplane.

Furthermore, the *blocked* predicate depends on an airplane. In the ADL version we simply iterate over all airplanes to check if a segment is blocked. This cannot be done in STRIPS so we add a single check for every airplane on the airport.

One problem still remains. STRIPS does not allow negated preconditions. Since we definitely need negated preconditions to test for unblocked and unoccupied segments we need to emulate their behavior. This is done by introducing negated versions of the *blocked* and *occupied* predicates:

```
(not_occupied ?s - segment)
(not_blocked ?s - segment ?a - airplane)
```

These predicates' logical value always needs to equal the negation of their positive counterparts. To achieve that, it is necessary to properly initialize them in the fact file and update them whenever their counterpart is updated. That means that if, for example, in the initial state a segment is not occupied a corresponding *not_occupied* predicate needs to be set true. Or, on the other hand, if we occupy a segment in an action's effect we also need to set the corresponding *not_occupied* predicate false.

This way we may easily replace all negated *blocked* or *occupied* preconditions.

Figure 3 shows an example of a non-durative STRIPS *pushback* action.

Now that we moved almost everything from the fact into the domain declaration it is obvious that we need a separate domain definition for each airport situation. The fact file merely defines the airplanes' starting state and the goal. All airport layout is implicitly defined by the move actions. Thus we can drop the *can-move*, *move-dir*, *can-pushback*, and *move-back-dir* predicates as they have been merged into the actions. The same is true for the *is-blocked* predicate. To make the pre-instantiating work, we need all segments, directions, and airplanes already defined in the domain description. So we declare them as constants instead of objects.

Just like we did with the ADL domain, the STRIPS version can be converted into a durative version. Since this is done the same way as with the ADL version we skip the details here.

### 3.3 Time Windows

The controllers of most large airports in the world only control their fore-field but not what is happening in airspace above the airport. That means they do not handle the landing of airplanes. So to further increase the realism of the airport domain we introduce this uncertainty by blocking the runway for the time of the landings. To simplify the procedure we just model these blocking *time-windows* and not the associated airplanes.

```
(:action pushback_seg_pp_0_60_seg_ppdoor_0_40_south_south_medium
 :parameters    (?a - airplane)
 :precondition  (and
                 (has-type ?a medium)
                 (is-pushing ?a)
                 (facing ?a south)
                 (at-segment ?a seg_pp_0_60)
                 (not_occupied seg_ppdoor_0_40)
                 (not_blocked seg_ppdoor_0_40 airplane_CFBEG)
                 (not_blocked seg_ppdoor_0_40 airplane_DAEWH)
                )
 :effect        (and
                 (not (occupied seg_pp_0_60))
                 (not_occupied seg_pp_0_60)
                 (not (blocked seg_pp_0_60 ?a))
                 (not_blocked seg_pp_0_60 ?a)
                 (not (at-segment ?a seg_pp_0_60))
                 (occupied seg_ppdoor_0_40)
                 (not (not_occupied seg_ppdoor_0_40))
                 (blocked seg_ppdoor_0_40 ?a)
                 (not (not_blocked seg_ppdoor_0_40 ?a))
                 (at-segment ?a seg_ppdoor_0_40)
                )
)
```

**Fig. 3.** A STRIPS pushback action: pushing a medium airplane from seg_pp_0_60 to seg_ppdoor_0_40 heading south

We have to model something like *segment s is blocked from time x to time y*. Clearly this can only be achieved with durative PDDL. The language for IPC-4, PDDL 2.2 [5], introduced *Timed Initial Literals*, which provide a very simple solution to the time-window problem. They allow the specification of a literal together with a time stamp at which the literal will become true. For example:

```
(at 119 (blocked seg_27 dummy_landing_airplane))
```

The above statement in the ":init" section of the fact file will make sure that segment seg_27 gets blocked at time 119. To unblock the segment after the landing we use a similar statement.

```
(at 119 (not (blocked seg_27 dummy_landing_airplane)))
```

Since the blocking predicate's second parameter is an airplane and we only simulate the landings but not the airplanes themselves, we use a dummy airplane for all landing actions which will be used nowhere else.

### 3.4 Optimization Criterion

We were not able to model the real optimization criterion of airport ground traffic control. The standard criterion in PDDL is to minimize the execution time, the *makespan*, of the plan. In our encoding of the domain this comes down to minimizing the arrival time of the last airplane. But the real objective is, as said above, to minimize the overall *summed up* travel time of all airplanes. There appears to be no good way of modeling this criterion in current PDDL. The difficulty lies in accessing the *waiting* times of the planes, i.e. the times at which they stay on a segment waiting for some other plane to pass. If one introduces an explicit waiting action then one must discretize time, in order to be able to tell the planner how long the plane is supposed to wait. For PDDL2.2, the introduction of a special fluent "current-time" was considered, returning the time point of its evaluation in the plan execution. Using such a "look on the clock", one could make each plane record its arrival time, and thus formulate the true optimization criterion in Airport. The IPC-4 organizing committee decided against the introduction of a "current-time" variable as it seems to be problematic from an algorithmic point of view (it implies a commitment to precise time points at planning time), and not relevant anywhere else but in the airport domain.

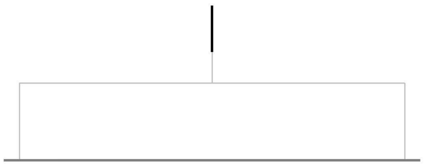
## 4 Results

To evaluate the performance of state-of-the-art planning systems in the Airport domain, we created five scaling example airports that we named "Minimal", "Mintoy", "Toy", "Half-MUC", and "MUC". The smallest of these airports is the smallest possible airport Astras can handle. The two largest airports correspond to one half of Munich Airport, MUC, respectively to the full MUC airport. Figure 4 shows sketches of the "Minimal" airport, and of the "MUC" airport. For each airport, we created a number of test examples that scaled by the number of airplanes to be moved, and by the number of time windows (if any). As indicated earlier in the paper, the test examples were generated by running an Astras simulation of the respective airport, then selecting various traffic situations during the simulation and putting them out in our different versions/encodings of the domain.

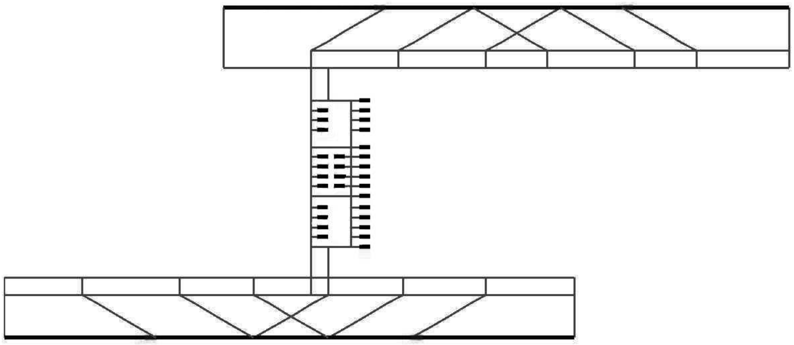
In our experiments, we ran the planners FF [9], IPP [10], MIPS [4], and LPG [7]. These planners are available online (programmed by one of the authors, in the cases of FF and IPP), and are suitable to represent the current state-of-the-art in sub-optimal and (step-)optimal fully automated planning, judging from the results of the international planning competitions in the previous years. FF was the winner of the 2000 competition, and is able to handle non-durational ADL representations. MIPS was awarded a 2nd place at both the 2000 and 2002 competitions, and handles durational STRIPS as well as timed initial literals. LPG was the winner of the 2002 competition, and handles durational STRIPS representations. IPP is the only optimal planner in our collection; it is based on the Graphplan [2] approach, handles non-durational ADL representations, and finds plans with a provably smallest number of parallel time steps.[1] IPP won the ADL track of the 1998 competition.

The planners were run under Linux on a 6 GB RAM computer with 4 Xeon 3.06 GHz CPUs. We report total runtimes as put out by the systems. Whenever a planning system ran out of memory on an example, the respective table entry shows a "—". We used a runtime cutoff of 10 minutes, and examples not solved within that time are

[1] Of course, finding step-optimal movement sequences is a long way away from the real optimization criterion in controlling airport ground traffic. Nevertheless, we find it interesting to see how far an optimal planner scales in the domain.



(a)



(b)

**Fig. 4.** Sketches of the "Minimal" (a) and "MUC" (b) airports. Park positions and runways are dark and taxiways are bright grey.

marked by ">>". We first give our results for the non-durational Airport domain, then for the durational domain without time windows, finally for the full durational domain with time windows.

### 4.1 Non-durational results

In the non-durational domain, two of our tested systems – namely FF and IPP – could handle the original non-compiled ADL domain. We ran FF and IPP on both the ADL and STRIPS encodings of the domain, and we ran LPG and MIPS on the STRIPS encoding only. Table 1 shows the results. We only report runtimes, not plan quality. From a real-life perspective, talking about plan quality – summed up travel time – does not make much sense in a non-durational setting where actions don't take any time.

Instances with a number of airplanes not solved by any of our planners are not shown in the table. We observe the following. First, in the ADL encoding no planner is able to scale to the MUC airports. The failure is due to a memory explosion in the pre-processing routines of both FF and IPP, which ground out all actions.[2] Strangely,

[2] While grounding out the actions comes very close to what we did in the STRIPS compilation, FF and IPP do it in a general way and fail during the creation of the necessary thousands of 1st order formulas (i.e., pointer tree structures).

| | | ADL | | STRIPS | | | |
|---|---|---|---|---|---|---|---|
| Airport | Nr. planes | FF | IPP | FF | IPP | LPG | MIPS |
| Minimal | 1 | 0.03 | 0.02 | 0.01 | 0.01 | 0.02 | 0.09 |
| Minimal | 2 | 0.05 | 0.04 | 0.01 | 0.02 | 0.09 | 0.11 |
| Mintoy | 1 | 0.39 | 0.26 | 0.02 | 0.05 | 0.08 | 0.25 |
| Mintoy | 2 | 0.59 | 0.42 | 0.01 | 0.21 | 0.09 | 0.30 |
| Mintoy | 3 | 0.85 | 0.86 | 0.02 | 0.57 | 0.16 | 0.67 |
| Mintoy | 4 | 1.09 | 5.56 | 0.76 | 9.54 | 17.45 | 15.12 |
| Toy | 1 | 0.55 | 0.35 | 0.02 | 0.08 | 0.18 | 0.30 |
| Toy | 2 | 0.79 | 0.55 | 0.03 | 0.24 | 0.48 | 0.36 |
| Toy | 3 | 1.38 | 1.05 | 0.06 | 0.38 | 0.62 | 0.68 |
| Toy | 4 | 1.50 | 3.52 | 3.04 | 4.63 | 0.82 | 37.12 |
| Toy | 5 | 1.80 | 239.19 | 19.30 | 190.76 | 9.84 | 38.64 |
| Toy | 6 | 2.52 | >> | 5.57 | >> | 22.17 | 164.31 |
| Toy | 7 | 6.60 | >> | 639.94 | >> | — | >> |
| Half-MUC | 2 | — | — | 0.25 | 8.96 | >> | 6.25 |
| Half-MUC | 3 | — | — | 0.37 | 18.73 | — | 13.51 |
| Half-MUC | 4 | — | — | 0.88 | 49.92 | — | — |
| Half-MUC | 5 | — | — | 1.34 | 61.14 | — | — |
| Half-MUC | 6 | — | — | 1.82 | 87.76 | — | — |
| Half-MUC | 7 | — | — | 3.59 | 131.13 | — | — |
| Half-MUC | 8 | — | — | 4.52 | 173.68 | — | — |
| Half-MUC | 9 | — | — | 5.60 | 240.01 | — | — |
| Half-MUC | 10 | — | — | 8.52 | — | — | — |
| MUC | 2 | — | — | 0.40 | 32.81 | — | — |
| MUC | 3 | — | — | 0.62 | 63.93 | — | — |
| MUC | 4 | — | — | 0.83 | 109.00 | — | — |
| MUC | 5 | — | — | 335.38 | 289.18 | — | — |

**Table 1.** Runtime results (in seconds) for the non-durational encodings of the Airport domain.

in those ADL instances that are feasible, FF is a lot more efficient than in the corresponding STRIPS instances. We investigated this phenomenon but could not find an explanation for it. It appears to be due to the internal ordering of the action instances. Our second observation is that all planners scale reasonably well in the smaller artificial airports, but run into trouble on the real-life sized MUC airports. On the negative side, LPG and MIPS solve hardly any of the MUC instances – it is unclear if this is due to the size of the search spaces, or to implementational difficulties with the pre-compiled STRIPS encoding. On the positive side, FF and IPP can solve Half-MUC instances even with many airplanes, and scale up to 5 planes on the real-life MUC airport. It is interesting to note that the optimal planner IPP appears to outperform the sub-optimal planners LPG and MIPS – a very unusual phenomenon in today's planning landscape. IPP is also slightly faster than FF in the largest of the solved examples (though it is significantly slower in the smaller ones). Altogether, it seems that planners are not too far away from real-life performance in the non-durational setting of this domain.

### 4.2 Durational results

From the real-life perspective, of course the durational version of the Airport domain is the more interesting one, particularly the version including time windows for the landing airplanes. Of our tested systems, only LPG and MIPS can handle durations, and only MIPS can handle the timed initial literals necessary to encode the time windows. Both LPG and MIPS handle only STRIPS representations so we could not run the ADL encoding. See the results for the STRIPS encodings in Table 2. We only report runtime. The found plans are all optimal with respect to the summed up overall travel time, see the discussion below.

| | | No Time Windows | | Time Windows |
|---|---|---|---|---|
| Airport | Nr. planes | LPG | MIPS | MIPS |
| Minimal | 1 | 0.03 | 0.15 | 0.18 |
| Minimal | 2 | 0.04 | 0.26 | 0.30 |
| Mintoy | 1 | 0.44 | 0.42 | 0.55 |
| Mintoy | 2 | 1.63 | 2.06 | 2.71 |
| Mintoy | 3 | 0.17 | 13.56 | 13.79 |
| Mintoy | 4 | >> | 74.88 | 19.45 |
| Toy | 1 | 0.10 | 0.50 | 1.16 |
| Toy | 2 | 0.37 | 2.41 | 13.77 |
| Toy | 3 | 0.24 | 13.65 | 83.59 |
| Toy | 4 | 2.76 | 101.51 | 102.68 |
| Toy | 5 | 3.46 | >> | >> |
| Toy | 6 | 5.48 | >> | >> |
| Toy | 7 | >> | >> | >> |
| Half-MUC | 2 | >> | 94.52 | 157.76 |
| Half-MUC | 3 | — | 405.19 | 497.93 |

**Table 2.** Runtime results (in seconds) for the durational STRIPS encoding of the Airport domain, with and without time windows.

As above, instances solved by none of the planners are not shown in the table. The obvious observation regarding runtime is that the durational domain is much harder for our planners than the non-durational domain – though as above it is unclear if LPG's and MIPS's inefficiency in the MUC airports is due to search complexity, or to the pre-compiled STRIPS encoding. LPG is generally more efficient than MIPS in the smaller airports, but fails completely on the larger Half-MUC airport.

Regarding plan quality, as said above already, LPG and MIPS find the optimal plans in all cases they can handle, i.e. they return the plans with the smallest possible summed up overall travel time. We checked that by hand. While this is a good result, one should keep two things in mind. First, LPG and MIPS do not know about the real optimization criterion so it is largely a matter of chance if or if not the plan they find is optimal with respect to that criterion. Second, the instances shown here – those cases that LPG and MIPS can handle – are very simple. With just a few airplanes, there is not much potential for (possibly) harmful interactions between the intended travel routes of these planes. In the above examples, often it is the case that there is just one non-redundant solution (a solution that does not leave planes standing around waiting without reason), and that this solution is the optimal one. Specifically this is the case in the four Half-MUC instances solved by MIPS.

We also wanted to run Acore, the (sub-optimal) scheduler integrated with Astras, on the above instances (i.e. in the respective traffic situations during the simulation with Astras), and compare the results with those of our planners. This turned out to not be feasible. In the small airports, there are a lot of parking conflicts, i.e. cases where an in-bound airplane is headed for a parking position that is occupied by an out-bound airplane. Such situations do rarely occur in reality (in fact, the flight schedules try to avoid these situations), and Acore can't handle them. In the larger MUC airports, on the other hand, our planners could not solve many instances. In the four Half-MUC instances solved by MIPS, Acore finds the trivially optimal solutions just like MIPS does. Generally, concerning runtime Acore is vastly superior to our planners. Acore can solve instances with 50 planes and more on Frankfurt airport, which is far beyond the scalability of the tested planning systems.

## 5 Conclusion

Our results show that today's PDDL planning systems are not powerful enough to handle the airport domain when it comes to real-life problems. Nonetheless we were able to solve small problems at a real airport in a simplified encoding without action durations, and MIPS solved a real-life encoding of one half of MUC airport with 3 airplanes within 10 minutes. This might still be far from the performance needed for real-life usage (at least ten airplanes need to be handled simultaneously in real-time) but it shows that PDDL has the potential for complex domains, and it shows that today's state-of-the-art planners are *a lot* closer to real-life applicability than they were some years ago.

The core problem in controlling the ground traffic on an airport is to resolve the *conflicts* that arise when two planes need to cross the same airport segment [8]. In our PDDL encoding, this core problem is hidden deep in the domain semantics, and it seems likely that the automated planners spend most of their runtime unawares of the core difficulties. For example, heuristic planners such as FF or MIPS recognize a (fatal kind of) conflict only when, during forward search, they find themselves in a world state that is deadlocked because two planes block each other's path to the destination. One can try to overcome these inefficiencies by not encoding in PDDL the physical airport, but only the conflicts and their possible solutions. Seeing if and how this is possible is a topic for future work. Another important open issue is how to best model the real optimization criterion in airport.

## References


1. Fahiem Bacchus. The AIPS'00 planning competition. *The AI Magazine*, 22(3):47–56, 2001.
2. Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):279–298, 1997.
3. Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.
4. Stefan Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research*, 20(195-238), 2003.
5. Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany, 2004.
6. Richard E. Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
7. A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.
8. Wolfgang Hatzack and Bernhard Nebel. Solving the operational traffic control problem. In A. Cesta, editor, *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'01)*, Toledo, Spain, September 2001. Springer-Verlag.

9. Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
10. Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, pages 273–285, Toulouse, France, September 1997. Springer-Verlag.
11. Drew McDermott et al. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee, 1998.
12. M.Fox and D.Long. The AIPS-2002 international planning competition. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*. AAAI Press, Menlo Park, 2002.
13. Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.
14. Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R. Brachman, H. J. Levesque, and R. Reiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 1st International Conference (KR-89)*, pages 324–331, Toronto, ON, May 1989. Morgan Kaufmann.
15. Sebastian Trüg, Jörg Hoffmann, and Bernhard Nebel. Applying automatic planners to airport ground-traffic control. Technical Report 199, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany, 2004.

# A The non-durative ADL Domain

```
(define (domain airport)

  (:requirements :equality :adl :typing)

  (:types airplane segment direction airplanetype)

  (:predicates
     (can-move ?s1 ?s2 - segment ?d - direction)
     (can-pushback ?s1 ?s2 - segment ?d - direction)
     (move-dir ?s1 ?s2 - segment ?d - direction)
     (move-back-dir ?s1 ?s2 - segment ?d - direction)
     (is-blocked ?s1 - segment ?t - airplanetype ?s2 - segment ?d - direction)
     (has-type ?a - airplane ?t - airplanetype)
     (at-segment ?a - airplane ?s - segment)
     (facing ?a - airplane ?d - direction)
     (occupied ?s - segment)
     (blocked ?s - segment ?a - airplane)
     (is-start-runway ?s - segment ?d - direction)
     (airborne ?a - airplane ?s - segment)
     (is-moving ?a - airplane)
     (is-pushing ?a - airplane)
     (is-parked ?a - airplane ?s - segment)
  )

  (:action move
  :parameters
     (?a - airplane ?t - airplanetype ?d1 - direction ?s1 ?s2 - segment ?d2 - direction)
  :precondition
     (and
         (has-type ?a ?t)
         (is-moving ?a)
         (not (= ?s1 ?s2))
         (facing ?a ?d1)
         (can-move ?s1 ?s2 ?d1)
```

```
        (move-dir ?s1 ?s2 ?d2)
        (at-segment ?a ?s1)
        (not (exists (?a1 - airplane) (and (not (= ?a1 ?a))
                         (blocked ?s2 ?a1))))
        (forall (?s - segment) (imply (and (is-blocked ?s ?t ?s2 ?d2)
                         (not (= ?s ?s1)))
                       (not (occupied ?s))
        ))
    )
:effect
    (and
        (occupied ?s2)
        (blocked ?s2 ?a)
        (not (occupied ?s1))
        (when (not (is-blocked ?s1 ?t ?s2 ?d2))
           (not (blocked ?s1 ?a)))
        (when (not (= ?d1 ?d2))
           (not (facing ?a ?d1)))
        (not (at-segment ?a ?s1))
        (forall (?s - segment) (when (is-blocked ?s ?t ?s2 ?d2)
                      (blocked ?s ?a)
        ))
        (forall (?s - segment) (when (and (is-blocked ?s ?t ?s1 ?d1)
                         (not (= ?s ?s2))
                         (not (is-blocked ?s ?t ?s2 ?d2))
                      )
                      (not (blocked ?s ?a))
        ))
       (at-segment ?a ?s2)
       (when (not (= ?d1 ?d2))
          (facing ?a ?d2))
    )
)

(:action pushback
:parameters
    (?a - airplane ?t - airplanetype ?d1 - direction ?s1 ?s2 - segment ?d2 - direction)
:precondition
    (and
       (has-type ?a ?t)
       (is-pushing ?a)
       (not (= ?s1 ?s2))
       (facing ?a ?d1)
       (can-pushback ?s1 ?s2 ?d1)
       (move-back-dir ?s1 ?s2 ?d2)
       (at-segment ?a ?s1)
       (not (exists (?a1 - airplane) (and (not (= ?a1 ?a))
                         (blocked ?s2 ?a1))))
       (forall (?s - segment) (imply (and (is-blocked ?s ?t ?s2 ?d2)
                         (not (= ?s ?s1)))
                      (not (occupied ?s))
       ))
    )
:effect
    (and
       (occupied ?s2)
       (blocked ?s2 ?a)
       (forall (?s - segment) (when (is-blocked ?s ?t ?s2 ?d2)
                      (blocked ?s ?a)
       ))
       (forall (?s - segment) (when (and (is-blocked ?s ?t ?s1 ?d1)
                        (not (= ?s ?s2))
                        (not (is-blocked ?s ?t ?s2 ?d2))
                      )
                      (not (blocked ?s ?a))
       ))
       (at-segment ?a ?s2)
```

```
      (when (not (= ?d1 ?d2))
         (facing ?a ?d2))
      (not (occupied ?s1))
      (when (not (is-blocked ?s1 ?t ?s2 ?d2))
         (not (blocked ?s1 ?a)))
      (when (not (= ?d1 ?d2))
         (not (facing ?a ?d1)))
      (not (at-segment ?a ?s1))
   )
)

(:action takeoff
:parameters
   (?a - airplane ?s - segment ?d - direction)
:precondition
   (and
      (at-segment ?a ?s)
      (facing ?a ?d)
      (is-start-runway ?s ?d)
   )
:effect
   (and
      (not (blocked ?s ?a))
      (not (occupied ?s))
      (not (at-segment ?a ?s))
      (airborne ?a ?s)
      (forall (?s1 - segment) (when (blocked ?s1 ?a)
                     (not (blocked ?s1 ?a)))
      )
   )
)

(:action park
:parameters
   (?a - airplane ?t - airplanetype ?s - segment ?d - direction)
:precondition
   (and
      (at-segment ?a ?s)
      (facing ?a ?d)
      (is-moving ?a)
   )
:effect
   (and
      (is-parked ?a ?s)
      (not (is-moving ?a))
      (forall (?ss - segment) (when (and (is-blocked ?ss ?t ?s ?d)
                        (not (= ?s ?ss)))
                     (not (blocked ?ss ?a)) )
      )
   )
)

(:action startup
:parameters
   (?a - airplane ?t - airplanetype ?s - segment ?d - direction)
:precondition
   (and
      (is-pushing ?a)
      (has-type ?a ?t)
      (at-segment ?a ?s)
      (facing ?a ?d)
      (forall (?s1 - segment)    (imply (and  (is-blocked ?s1 ?t ?s ?d)
                            (not (= ?s ?s1)))
                        (not (occupied ?s1))
       ))
   )
```

```
  :effect
     (and
         (not (is-pushing ?a))
         (is-moving ?a)
         (forall (?s1 - segment)    (when  (is-blocked ?s1 ?t ?s ?d)
                           (blocked ?s1 ?a)
         ))
     )
  )
)
```

# B The durative ADL domain

```
(define (domain airport_durative)

  (:requirements :durative-actions :equality :adl :typing)

  (:types airplane segment direction airplanetype)

  (:predicates
     (can-move ?s1 ?s2 - segment ?d - direction)
     (can-pushback ?s1 ?s2 - segment ?d - direction)
     (move-dir ?s1 ?s2 - segment ?d - direction)
     (move-back-dir ?s1 ?s2 - segment ?d - direction)
     (is-blocked ?s1 - segment ?t - airplanetype ?s2 - segment ?d - direction)
     (has-type ?a - airplane ?t - airplanetype)
     (at-segment ?a - airplane ?s - segment)
     (facing ?a - airplane ?d - direction)
     (occupied ?s - segment)
     (blocked ?s - segment ?a - airplane)
     (is-start-runway ?s - segment ?d - direction)
     (airborne ?a - airplane ?s - segment)
     (is-moving ?a - airplane)
     (is-pushing ?a - airplane)
     (is-parked ?a - airplane ?s - segment)
  )

  (:functions
     (length ?s - segment)
     (engines ?a - airplane)
  )

  (:durative-action move
  :parameters
     (?a - airplane ?t - airplanetype ?d1 - direction ?s1 ?s2 - segment ?d2 - direction)
  :duration
     (= ?duration (/ (length ?s1) 30))
  :condition
     (and
         (over all (has-type ?a ?t))
         (over all (is-moving ?a))
         (over all (not (= ?s1 ?s2)))
         (at start (facing ?a ?d1))
         (over all (can-move ?s1 ?s2 ?d1))
         (over all (move-dir ?s1 ?s2 ?d2))
         (at start (at-segment ?a ?s1))
         (over all (not (exists (?a1 - airplane)    (and  (not (= ?a1 ?a))
                                    (blocked ?s2 ?a1))))
         )
         (over all    (forall (?s - segment) (imply (and  (is-blocked ?s ?t ?s2 ?d2)
                                    (not (= ?s ?s1)))
                                (not (occupied ?s))
                           )
```

```
                )
            )
        )
    :effect
        (and
            (at start (occupied ?s2))
            (at start (blocked ?s2 ?a))
            (at end (not (occupied ?s1)))
            (at end (when  (not (is-blocked ?s1 ?t ?s2 ?d2))
                    (not (blocked ?s1 ?a)))
            )
            (at end (when  (not (= ?d1 ?d2))
                    (not (facing ?a ?d1)))
            )
            (at end (not (at-segment ?a ?s1)))
            (at end     (forall (?s - segment) (when  (is-blocked ?s ?t ?s2 ?d2)
                                (blocked ?s ?a)
                            )
                    )
            )
            (at end     (forall (?s - segment) (when  (and  (is-blocked ?s ?t ?s1 ?d1)
                                    (not (= ?s ?s2))
                                    (not (is-blocked ?s ?t ?s2 ?d2))
                                )
                                (not (blocked ?s ?a))
                            )
                    )
            )
            (at end (at ?a ?s2))
            (at end (when  (not (= ?d1 ?d2))
                    (facing ?a ?d2))
            )
        )
    )

    (:durative-action pushback
    :parameters
        (?a - airplane ?t - airplanetype ?d1 - direction ?s1 ?s2 - segment ?d2 - direction)
    :duration
        (= ?duration (/ (length ?s1) 5))
    :condition
        (and
            (over all (has-type ?a ?t))
            (over all (is-pushing ?a))
            (over all (not (= ?s1 ?s2)))
            (at start (facing ?a ?d1))
            (over all (can-pushback ?s1 ?s2 ?d1))
            (over all (move-back-dir ?s1 ?s2 ?d2))
            (at start (at-segment ?a ?s1))
            (over all    (not  (exists (?a1 - airplane)    (and  (not (= ?a1 ?a))
                                    (blocked ?s2 ?a1)))
                    )
            )
            (over all    (forall (?s - segment) (imply (and  (is-blocked ?s ?t ?s2 ?d2)
                                    (not (= ?s ?s1)))
                                (not (occupied ?s))
                            )
                    )
            )
        )
    :effect
        (and
            (at start (occupied ?s2))
            (at start (blocked ?s2 ?a))
            (at end (forall (?s - segment) (when  (is-blocked ?s ?t ?s2 ?d2)
                            (blocked ?s ?a)
                        )
```

```
            )
        )
        (at end (forall (?s - segment) (when  (and  (is-blocked ?s ?t ?s1 ?d1)
                                (not (= ?s ?s2))
                                (not (is-blocked ?s ?t ?s2 ?d2))
                            )
                            (not (blocked ?s ?a))
                        )
            )
        )
        (at end (at-segment ?a ?s2))
        (at end (when  (not (= ?d1 ?d2))
                (facing ?a ?d2))
        )
        (at end (not (occupied ?s1)))
        (at end (when  (not (is-blocked ?s1 ?t ?s2 ?d2))
                (not (blocked ?s1 ?a)))
        )
        (at end (when  (not (= ?d1 ?d2))
                (not (facing ?a ?d1)))
        )
        (at end (not (at-segment ?a ?s1)))
    )
)

(:durative-action takeoff
:parameters (?a - airplane ?s - segment ?d - direction)
:duration  (= ?duration 30)
:condition
    (and
        (at start (at-segment ?a ?s))
        (at start (facing ?a ?d))
        (over all (is-start-runway ?s ?d))
    )
:effect
    (and
        (at end (not (blocked ?s ?a)))
        (at end (not (occupied ?s)))
        (at end (not (at-segment ?a ?s)))
        (at end (airborne ?a ?s))
        (at end (forall (?s1 - segment) (when  (blocked ?s1 ?a)
                            (not (blocked ?s1 ?a))
                        )
            )
        )
    )
)

(:durative-action park
:parameters
    (?a - airplane ?t - airplanetype ?s - segment ?d - direction)
:duration
    (= ?duration 40)
:condition
    (and
        (over all (at-segment ?a ?s))
        (over all (facing ?a ?d))
        (at start (is-moving ?a))
    )
:effect
    (and
        (at end (is-parked ?a ?s))
        (at end (not (is-moving ?a)))
        (at end (forall (?ss - segment) (when  (and  (is-blocked ?ss ?t ?s ?d)
                            (not (= ?s ?ss))
                        )
                    (not (blocked ?ss ?a))
```

```
                    )
            ))
        )
    )

    (:durative-action startup
    :parameters
        (?a - airplane ?t - airplanetype ?s - segment ?d - direction)
    :duration
        (= ?duration (* 60 (engines ?a) ) )
    :condition
        (and
            (at start (is-pushing ?a))
            (over all (has-type ?a ?t))
            (over all (at-segment ?a ?s))
            (over all (facing ?a ?d))
            (over all (forall (?s1 - segment)   (imply (and  (is-blocked ?s1 ?t ?s ?d)
                                          (not (= ?s ?s1)))
                                      (not (occupied ?s1))
            )))
        )
    :effect
        (and
            (at end (not (is-pushing ?a)))
            (at end (is-moving ?a))
            (at end (forall (?s1 - segment) (when  (is-blocked ?s1 ?t ?s ?d)
                                   (blocked ?s1 ?a)
            )))
        )
    )
)
```