Master's Thesis

Domain Knowledge in TALplanner

by

Martin Magnusson

LiTH-IDA-Ex-02/104

2003-01-07

Supervisor: Jonas Kvarnström

Examiner: Patrick Doherty

Abstract

Planning with domain knowledge is a relatively new and growing research area that allows tackling much larger planning problems than were previously possible. This is achieved by allowing the encoding of domain knowledge as search control rules or heuristics, reducing the search space and guiding the search. This thesis describes the process of creating control rules in a number of different planning domains for a planner that makes use of domain knowledge, TALplanner. The domains were part of the 2002 International Planning Competition in which TALplanner participated. Though no prizes were awarded to TALplanner, it performed very well in competition with the other planners. Also discussed are the challenges met and the modifications made to the planner in order to perform efficiently in the domains and comply with all contest rules.

Contents

List of Figures

Chapter 1

Introduction

This chapter explains the background of the thesis, its purpose, limitations, and structure.

1.1 Background

In 1997, Linköping University received funding for the WITAS [14] project with the long-term goal to develop a fully autonomous unmanned helicopter. As the helicopter is supposed to make plans by itself, one part of the system is a planner, TALplanner, developed by Jonas Kvarnström and Patrick Doherty. TALplanner participated in the 2000 International Planning Competition [3] at the Artificial Intelligence Planning and Scheduling conference [16] and won first prize in the hand-tailored track. The competition is a biennial event and it was decided that TALplanner would enter again in 2002. This thesis describes the preparations for and results of that competition.

1.2 Purpose

The goals that we have set up for this thesis are the following:

1. Give an introduction to TAL and TALplanner.
2. Present the domains from the 2002 planning competition.
3. Describe the modeling of these domains.
4. Describe the changes and extensions made to TALplanner in order to perform well in the domains.
5. Present the competition results.

1.3 Limitations

This thesis focuses upon planning with domain knowledge and some of the content is applicable to all planners that can make use of such knowledge but much of it is limited to TALplanner, which is the planner used for all the contest planning problems. Control rules that are presented for the problem domains often deal with performance issues only relevant in the context of TALplanner's implementation details. Control rules are also not the only possible way to specify domain knowledge. For example, some planners support heuristic rules but no such rules have been included in the thesis.

Consistent with the nature of domain dependent knowledge, part of the knowledge formalized as control rules and presented in the thesis is inapplicable to other domains than those it was specifically developed for. This is not true for all the knowledge since many planning domains contain similar objects and actions. For instance, logistical problems are common and present similar complications and solutions.

1.4 Structure of the Thesis

The thesis is structured as follows:

Chapter 2 introduces planning and some of the terminology used by the planning community.

Chapter 3 describes TALplanner, its background, and briefly its workings and the syntax of its use.

Chapter 4 lists each contest planning domain TALplanner participated in, explaining the control rules developed and the reasoning behind them.

Chapter 5 gives a short description of all the modifications and additions made to TALplanner to enable it to perform efficiently in the planning domains.

Chapter 6 presents the competition results with a number of graphs, making comparison between the planners competing in the contest straightforward.

Chapter 7 discusses the insights that have been gained during the work on this thesis.

Appendix A defines some commonly used terminology.

Appendix B contains the complete domain definitions.

Chapter 2

Planning

This chapter will introduce the basics of planning and some of the terminology used by the planning community. Also discussed is the important distinction between domain independent and domain dependent planning.

2.1 Planning

Planning is the process of finding a sequence or set of actions that change parts of a world from some initial state to a goal state. A planner is a computer program that uses some form of search to look for such a sequence or set.

The state of the world is represented in some formal way, often by a set of predicates that express which facts about objects and environment are true. By the closed world assumption, facts that are not explicitly specified as true are assumed to be false. This convention avoids the trouble of enumerating all possible predicates in every state and is intuitively appealing. For example, if one was asked to describe the top desk drawer, one might enumerate its contents but would certainly not continue, enumerating every conceivable object that is *not* in the drawer.

The actions that transform states are specified in the form of operators. An operator can be split into two parts: the preconditions, which limit the applicability of the operator to certain states, and the effects, or post conditions, which define the changes that are made to the state when the operator is applied.

In its simplest form, planning is done in an accessible and deterministic environment. This means that the planner can check if some predicate is true and treat the result as a fact, and that the outcome of applying an operator can always be determined in advance. This can be put into contrast with the real world where no foolproof tests of facts exist and the outcome of taking action is always uncertain. Note that determinism does not rule out actions with conditional effects that depend on the environment since finding which of these effects will actually happen is only a matter of testing the relevant predicates in the, accessible, environment in which the action was performed. Even in this restricted and somewhat unrealistic environment, planning is *not* a simple task.

For research purposes a large number of problem domains that fulfill the constraints described above have been created, possibly the most (in)famous of these being the blocks world [15]. To make the concepts above more tangible this problem is used in an example.

The blocks world domain consists of a set of blocks on a table. The blocks can be stacked on top of each other, but only in straight towers. A robot arm can pick up and put down one block at a time but not lift a block which lies underneath another block without first moving that one. The table is sufficiently large to make room for any number of blocks.

We can describe the domain using the following constructs. A number of constants, A, B, C, …, are introduced to represent the blocks and a number of variables, *x*, *y*, *z*, *v*, *w*, are used to refer to any block or the table. The world state is expressed using the predicate On(*x*, *y*), meaning that block *x* is directly on top off *y*, where *y* is either another block or the table. Finally, an operator Move(*x*, *y*, *z*) is defined with the meaning move block *x* from *y* to *z*.

Move(*x*, *y*, *z*)
Preconditions: On(*x*, *y*)
Not exists *v* such that On(*v*, *x*)
Not exists *w* such that both On(*w*, *z*) and *z* != table
Effects: Not On(*x*, *y*)
On(*x*, *z*)

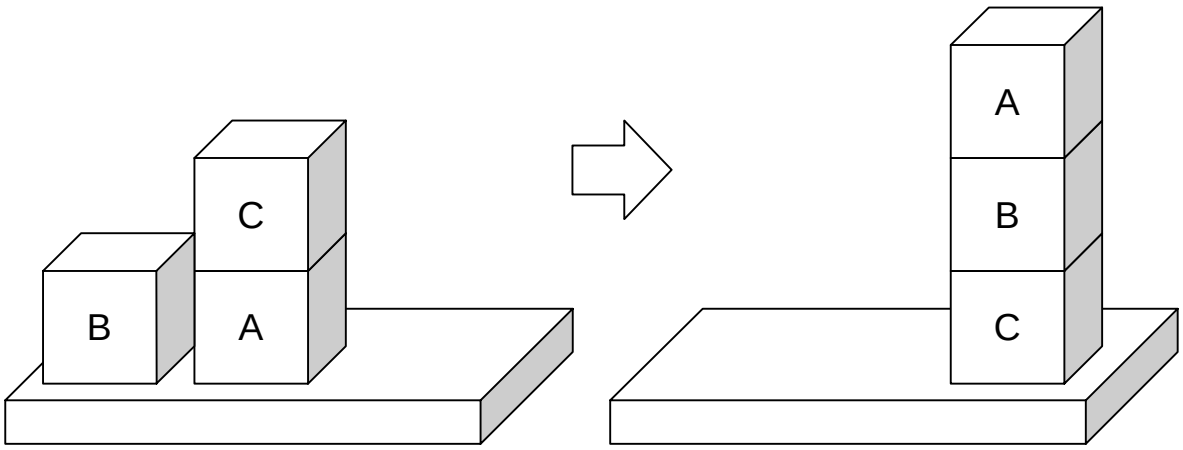

Figure 2.1: A classic blocks world problem.

Figure 2.1 is a graphical representation of a small planning problem. There is an initial state and a goal state, and the task is to find a sequence of Move actions that rearranges the blocks from the initial state into the goal state. Using the above representation scheme and, consistent with the closed world assumption, specifying only positive facts, the initial state is encoded as,

On(A, Table)
On(B, Table)
On(C, A)

The goal state is similarly described as,

On(A, B)
On(B, C)
On(C, Table)

The following action sequence would constitute one solution to the problem:

1. Move(C, A, Table)
2. Move(B, Table, C)
3. Move(A, Table, B)

2.2 Different Approaches to Knowledge in Planning

Much of the early work in planning was directed towards solving problems from any domain using a minimal amount of knowledge about the domain. This approach only requires a formalization of the domain operators in the planner's input language but severely limits the planner's ability to solve large problems. The search space grows exponentially as the problem size increases and it becomes necessary to prune large parts of the search tree or direct the search in some other way to find a solution.

Minimal knowledge, or fully automated planning as it is sometimes called, is one extreme and domain dependent planning is the other. A domain dependent planner is limited to one planning domain or a set of closely related domains. Knowledge about the domain is present not just in the input to the planner but in the actual implementation and algorithms of the planner. Domain dependent planners can be very efficient, but the downside is obvious. They are only special-purpose software and modifying such a planner to a new application area may require changes in its entire architecture and involves a lot of work.

TALplanner covers the middle ground between the two extremes, belonging to a group of planners that might be termed hand-tailored planners. It is domain independent and can create plans with only a minimum of domain knowledge but it also lets the user provide additional information about the domain to help control and limit the search for solutions. Very refined control knowledge may be added, completely eliminating the need for search and always forcing the planner to choose an action that will be part of the final solution. Such strict control brings TALplanner closer to domain dependence but is not required. Any form of knowledge that will guide or control the search increases the size of the problems that are possible to solve. An example of such knowledge for the blocks world example may be an instruction not to move blocks that are already in a goal state configuration. When confronted with a real world domain, there are probably experts in the field who already have a lot of knowledge of how to solve problems in the domain and it makes sense to be able to use that knowledge when modeling it. Furthermore, creating a planner that supports the addition of hand tailored domain knowledge allows experimentation with using various forms of knowledge that might later be generated automatically.

There are of course downsides to using domain specific knowledge in planning. First of all, finding and encoding the knowledge creates a significant amount of extra work. Secondly, such knowledge is not always intuitive, or even available at all. Finally, there is always a risk of providing erroneous information that will actually hinder or prevent the planner from solving certain problems.

Chapter 3

TALplanner

The planner used throughout this thesis is TALplanner [18] [17][19][2], developed by Jonas Kvarnström and Patrick Doherty. Its history started in 1999 as part of the WITAS Unmanned Aerial Vehicle project [14], but has since then grown into a complete standalone planning project.

This chapter introduces TAL, the formal basis for the planner, TALplanner and some of its inner workings, and finally describes the function of control rules, the development of such rules being the topic of Chapter 4.

3.1 TAL

TAL, an acronym for Temporal Action Logics, is a formalism that uses first and second order logic to describe actions and change in a world model. It can be used to reason about actions that have duration, actions that are performed concurrently with other actions, context-dependent actions and much more. TAL's expressiveness provides a suitable formal basis for a planner and is indeed used by TALplanner as such. It is not necessary to have a formal basis at all to create a planner, and many planners do not, but it does make it possible to prove correctness of the plans generated if provided. More information about TAL can be found in Doherty et al. [1].

3.2 TALplanner

As TALplanner is intended to be used in the context of an unmanned helicopter, certain constraints need to be met. Especially, a plan must be found within some limited time period, but the problem domain is of limited scope, opening up the possibility of using domain specific knowledge to help achieve any time constraints. A survey of the planning research area revealed only one existing planner that was of interest and fulfilled the constraints. That planner was TLPlan, developed by Bacchus and Kabanza [9]. TLPlan served as inspiration and together with the experience with TAL led to the development of a forward chaining logic-based planner where domain knowledge is expressed as logical formulas, controlling the search for solution plans. Forward chaining planners start from the initial world state, adding actions to the plan until the goal is reached. Many other planning algorithms have been developed but forward chaining has several advantages, among which are ease of world state representation and use of complex operators. The biggest disadvantage is that no goal directedness exists and consequently quite elaborate control may have to be imposed on the search to make the planning process efficient.

3.2.1 Pruning Constraints in TALplanner

The nature of the depth-first search algorithm used by TALplanner essentially means that, without any control rules, the planner will try all possible instantiations of all the operators in the order in which they were defined. In a sense, the planner adds an action, not because it is necessary, but because it is possible. It is therefore essential to provide domain specific control rules that guide the search in order to achieve any sort of reasonable efficiency.

TALplanner analyzes the control rules and creates pruning constraints, logic formulas that must hold in any partial plan, i.e. in any node in the search tree. If the pruning constraint does not hold, the state node can be pruned, and with it, the entire branch of the search tree that would stem from that node. The result is a drastic reduction of the search space, depending on the quality of the control rules, enabling the depth-first search algorithm, which would ordinarily be totally lost among the vast number of possible plans, to find a solution. More information about the generation of pruning constraints is available in [4].

3.2.2 Basic Algorithm

The search algorithm used by TALplanner is a standard depth-first search, although other strategies, like breadth first search or iterative deepening depth-first search, are possible and easily implemented through the plug in-like code architecture. The steps in the algorithm can be listed using pseudo code as follows:

1. **procedure** TALplan(state)
2. **if** the control rules are satisfied in the state **then**
3. **if** the state is a goal state, **return** the state.

4. **if** the state does not constitute a cycle **then**
5. **for** every action that is applicable in the state **do**
6. **call** TALplan recursively with a new state, which results from applying the action to the current state, as an argument.
7. **return** failure.

A cycle is present if the state has already been visited earlier in the search and cycle checking is necessary in depth-first searches to prevent the planner from getting stuck, repeatedly adding actions that cancel each others effects out.

Note that even if the return value is failure, the problem might still be solvable. One or more of the control rules may be too strict and exclude the branches containing the solution or solutions from the search.

Presenting the basic search algorithm merely scratches the surface of what TALplanner is. The main body of work lies in all the optimizations that are done on the logical formulas and the solutions of all the representational issues encountered. As this lies outside the scope of this thesis, the reader is referred to Kvarnström [4].

3.2.3 Input and Output Language

The syntax of operator definitions is best explained with an example. An operator `drive`, which drives an available taxi between two locations, could be declared as follows:

```
#operator drive(taxi, location1, location2)
 :at t
 :precond    [t] at(taxi, location1) &
             [t] available(taxi)
 :context
   :effects  [+1] at(taxi, location1) := false,
             [+10] at(taxi, location2) := true
```

At instantiation, when the operator is added as an action to the plan, the variable `t` is bound to the current time point. The preconditions, represented as a single temporal logic formula, must hold for the operator to be applicable. In order to drive the taxi from `location1` to `location2` at time point t, it must be both at `location1` and available at time point t. Following the precondition is a list of contexts, each of which can have their own preconditions and effects. The `drive` operator has only one context and one list of effects. Consequently it does not need additional preconditions in that context and always has the same effects when applied. At time point t + 1 the taxi has left `location1`, and at t + 10 it has arrived at `location2`.

Continuing with an example control rule helps explain the control rule syntax.

```
#control :name "don’t-drive-to-deserted-places"
         forall t, taxi, location [
                [t] !at(taxi, location) &
                ([t+1] at(taxi, location)) ->
                exists person [
                       [t] at(person, location) ] ]
```

The new control rule is given a name and then defined by a single logical formula. The formula first quantifies universally over time points, taxis and locations and then, by means of an implication and an existential quantification over persons, makes sure that if a taxi arrives at a location, it must be the case that at least one potential customer is there.

The output plan can be printed in two different formats: a TALplanner native format and a Planning Domain Definition Language (PDDL) [7][8] format. Here we shall only be concerned with the latter, once again providing an example.

```
0 : (drive taxi1 citysquare airport) [ 10 ]
10 : (pickup person1 airport) [ 2 ]
12 : (drive taxi1 airport suburb) [ 10 ]
22 : (dropoff person1 suburb) [ 4 ]
;; Plan length 4, maxtime 26
```

Assuming appropriate definitions of `pickup` and `dropoff`, the plan might be a solution to some simple planning problem. Beginning each row is the time point at which the action was applied. The instantiated operator is presented in a LISP-like format for PDDL compatibility and the rows end with the duration of the operator inside brackets. For convenience, the last row displays the number of steps in the plan and the time point at which the goals were achieved.

Chapter 4

IPC02

4.1 History

AIPS is a conference on Artificial Intelligence Planning and Scheduling, which is held every other year since 1998. The conference hosts the International Planning Competition (IPC) in which TALplanner has previously entered and received awards.

Since the year 2000, the IPC includes both fully automated and hand-tailored planners (see section 2.2). A set of problem domains and two sets of problems for each domain are given and a deadline for handing in the solutions is fixed. Hand-tailored planners are typically much faster and can solve larger problems than the fully automatic planners. The second sets of problems take this into account, as they are larger versions of the problems in the first set. The domains are formalized in the Planning Domain Definition Language (PDDL) [7][8]. PDDL is a currently active and evolving attempt to standardize planning problems and makes it possible for different planners to compete against each other by directly supporting PDDL or translating the PDDL definitions to their respective input language and back again when a plan has been found. The competitors solve as many problems as they can before the deadline while timing the planner's execution. This creates a large set of data to judge the performance by, including time spent solving the problems, length of the plans generated and domain specific criteria, e.g. the amount of resources, like fuel, spent in each problem's solution plan.

Note that the setup of the competition may change in the future since the group of people responsible for organizing the competition is not fixed but changes each year the conference is held.

The latest AIPS conference was held in France 2002. Responsible for the 2002 competition (IPC02) [3] were Derek Long and Maria Fox. This chapter introduces the 2002 domains, explains how we modeled each of them and discusses any particular difficulties encountered.

4.2 Domains

All competition domains except one have several versions of varying complexity. The first and simplest is the “STRIPS” version, which is compatible with the STRIPS planning formalism [6] and therefore also limited to the expressivity of STRIPS, which among other things excludes actions that have a duration of more than one time step. The second is “SimpleTime” where actions have a specified constant duration. In the “Timed” version the duration of actions can be dependent on the problem instance being solved, e.g. driving a vehicle takes time proportional to the distance covered. “Numeric” versions have no durative actions but instead introduce numeric constraints like limiting the loading capacity of a vehicle. Finally some domains have a fifth “Complex” version that is a combination of Timed and Numeric.

Even though TALplanner is expressive enough to attempt the Numeric and Complex problems, we decided to take part only in the STRIPS, SimpleTime and Timed domains and maximize its performance there. The decision ensured that the deadline was met but was probably suboptimal from a strategic point of view since, after the contest, it was revealed that an important criterion in the final judgment was overall problem coverage.

Concurrent planning, where several actions can be performed simultaneously, was used for all domains except the largest one, which used sequential planning, where one action has to be completed before the next one can begin.

The rest of this chapter contains descriptions of each domain, the modeling of it to allow efficient planning using TALplanner and which obstacles we encountered. Many of the difficulties are not specific for a single domain but appear in several or all of the domains. These problems are described in depth when first mentioned and then skipped over to avoid repetition. The sections are therefore not completely independent and should preferably be read in the order they appear.

The complete domain definitions are quite long and are placed in Appendix B for reference. Smaller fragments of the formalizations are inserted in the text to illustrate the concepts and methods used.

4.3 ZenoTravel

The ZenoTravel domain is based on a domain created to illustrate the capabilities of the Zeno planner [5]. The task is to fly people between different towns using a set of aircraft.

The complete domain and control rule definitions for the domain are available in appendices B.1, B.2 and B.3 and can be used as reference while reading the following description.

4.3.1 Description

Only five actions are available. Persons may `board` and `debark` aircraft and aircraft may `fly`, `zoom` and `refuel`. There are no restrictions on how many people a plane can carry. Flying and zooming are equivalent except zooming is faster and uses more fuel.

Figure 4.1 shows an example problem. Part of the solution would involve one of the planes flying to `city0`, `person1` boarding the plane, the plane flying to `city3` and `person1` disembarking.

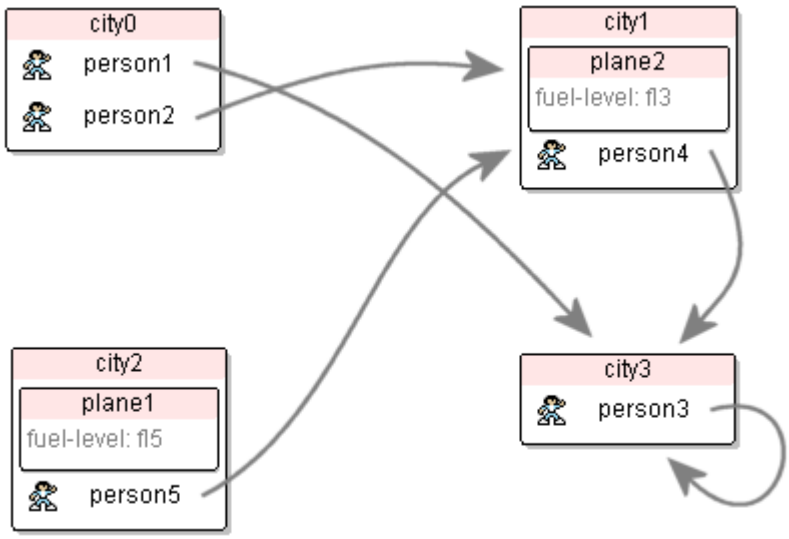

Figure 4.1: A simple sample problem from IPC02 with arrows pointing out goal locations. A solution is a plan that puts `person1`, `person3` and `person4` in `city3` and `person2` and `person5` in `city1`.

4.3.2 Control

The first and most obvious fact we notice is that people who are already at their destinations do not need to do anything. They are prevented from doing anything by adding the following control rule or something equivalent:

```
#control :name "only-board-when-necessary"
     forall t, person, aircraft [
          [t] !in(person, aircraft) &
          ([t+1] in(person, aircraft)) ->
          exists city [
               [t] at(person, city) &
               goal(!at(person, city)) ]]
```

The rule says that at all times for all persons and aircraft, if the person boards the aircraft he must be at one city and have a goal to be at another city.

Think of `[t] at(person, city) & goal(!at(person, city))` as short for:

```
exists city1, city2 [
     [t] at(person, city1) &
     goal(at(person, city2)) &
     city1 != city2 ]
```

The statement `goal(!at(person, city))` is only true if `!at(person, city)` is true in all goal states. It must be the case that there is a goal forcing the person to be somewhere else, otherwise there would be goal states in which `at(person, city)` was true and its negation false.

In reality, planes usually follow predefined routes, but in this simplified model, planes can fly directly to any other city. A natural extension of the rule is therefore that people only disembark at their destinations instead of getting off the plane in random cities.

```
#control :name "only-debark-when-in-goal-city"
     forall t, person, aircraft [
          [t] in(person, aircraft) ->
          ([t+1] in(person, aircraft)) |
          exists city [
               [t] at(aircraft, city) &
               goal(at(person, city)) ] ]
```

These two rules help people behave rationally but the aircraft still fly wherever they can. Three reasons exist for a plane to visit a city: one of the goals asserts that the aircraft must end up in the city when the plan is complete, there is a person already in the aircraft that wants to go to the city, or there is a person in the city that wants to leave. The following rule formalizes these three intuitions:

```
#control :name "planes-always-fly-to-goal"
     forall t, aircraft, city [
          [t] at(aircraft, city) ->
          ([t+1] at(aircraft, city)) |
          exists city2 [
               city2 != city &
               ([t+1] at(aircraft, city2)) &
               (goal(at(aircraft, city2)) |
                exists person [
                    [t] in(person, aircraft) &
                    goal(at(person, city2)) ] |
                exists person [
                    [t] at(person, city2) &
                    goal(!at(person, city2)) ]) ] ]
```

The first criterion proves to be too admissible. A goal stating that the aircraft must be in a certain city, its goal city, should not really be of concern until all the passengers have arrived at their destinations. There is no point visiting the goal city in the middle of the plan, if not to pick up or drop off passengers. We define a new feature to help decide when to fly the plane to its goal city:

```
#define [t] all-persons-at-their-destinations:
     forall person, city [
          goal(at(person, city)) -> [t] at(person, city) ]
```

The new feature will become true when all persons that have a goal city have arrived at it. Adding it as an extra requirement to the first case in the previous rule solves the problem.

```
#control :name "planes-always-fly-to-goal"
     forall t, aircraft, city [
          [t] at(aircraft, city) ->
          ([t+1] at(aircraft, city)) |
          exists city2 [
               city2 != city &
               ([t+1] at(aircraft, city2)) &
               ((goal(at(aircraft, city2)) &
                [t] all-persons-at-their-destinations) |
                exists person [
                    [t] in(person, aircraft) &
                    goal(at(person, city2)) ] |
                exists person [
                    [t] at(person, city2) &
                    goal(!at(person, city2)) ]) ] ]
```

With these simple rules TALplanner solves all the ZenoTravel contest problems quickly. Although the solutions are not terribly inefficient, they can be improved by spotting several problems. For example, many plans involve flying all available aircraft to pick up one person.

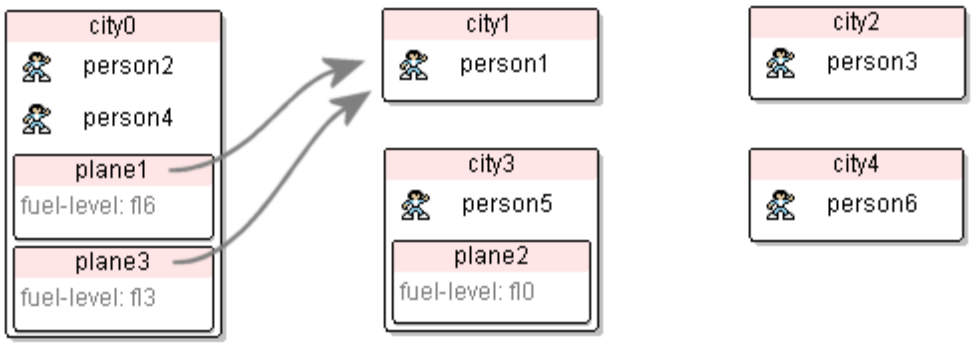

Figure 4.2: Contest problem with the beginning of a plan marked.

Figure 4.2 illustrates this problem. The first steps of the generated solution plan is:

```
0 : (fly plane1 city0 city1 fl6 fl5) [ 1 ]
0 : (fly plane3 city0 city1 fl3 fl2) [ 1 ]
```

But only one of the planes can make itself useful by picking up the lone passenger. This is a general problem when planning concurrently using a depth-first search strategy such as TALplanner's. The planner checks the state at time 0 and looks for actions that are possible to perform. An action, fly `plane1` from `city0` to `city1` is added to the plan, but the action does not change the initial state so it is not possible to write a control rule that in the initial state forbids flying `plane3` to the same location without reference to the next time step, when `plane1` has arrived at `city1`. That state is still in the future and is unknown since more actions may be added at time 0 and affect the state at time 1. The rule would be evaluated at time 0 when the planner tries to fly `plane3` but instead of canceling the action, the action will be added and the control rule queued and evaluated at a later time when the future state is fully determined. The planner will continue to add actions until the queued rule, at some later time, forces it to backtrack. Depending on the number of actions added after the flying of `plane3`, the time it takes for the planner to realize that this action was not allowed could be very long.

To prevent this problem we added a `committed` macro. When the planner decides to fly `plane1` to `city1` it registers the fact `at(plane1, city1)` as committed to be true in the next state. We can now add the following check to the last case of the `planes-always-fly-to-goal` rule that permits flying to a city to pick up someone:

```
!exists aircraft2 [
     aircraft2 != aircraft &
     $committed(t+1, at(aircraft2, city2), true) ]
```

A plane is not allowed to fly to a city in order to pick up a passenger if there already exists another plane that has decided to go there.

One final important and peculiar discovery is made when looking at the given operator definitions for the domain. The only difference between the `fly` and the `zoom` operators is that `zoom` uses twice as much fuel. Zooming is not faster than flying in the STRIPS version of the domain where all actions take exactly one time step. The `zoom` operator was thus commented out to make sure it is not used.

4.3.3 SimpleTime

The only difference in the given PDDL specification between the SimpleTime and the STRIPS version is that instead of all actions taking only one time step to perform, they now have a constant non-zero duration and some of the preconditions must hold throughout the action's entire execution period. However, this translates into a number of changes in the TALplanner formalization to enable almost as fast planning as with the STRIPS version.

If we try using the same definitions, with durations added and the effects modified to occur at the end of the duration, we run into problems.

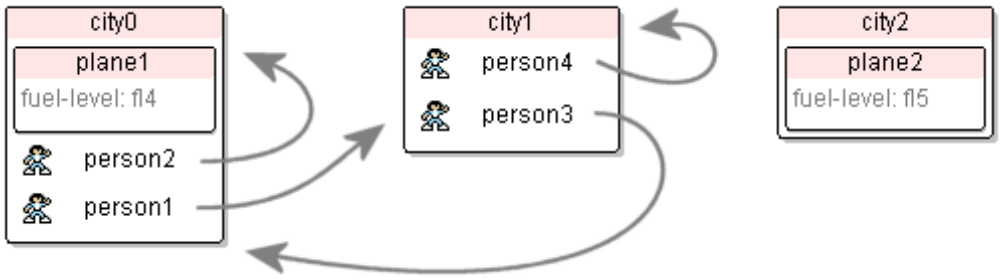

Figure 4.3: Contest problem with peoples destinations displayed.

Let us look at the example in Figure 4.3. The planner starts by adding the following actions to the plan:

```
[0,20] board(person1, plane1, city0)
[0,20] board(person2, plane1, city0)
[0,100] fly(plane2, city2, city0, fl5, fl4)
Failed queued constraint from planes-always-fly-to-goal
```

It is unable to continue due to the `planes-always-fly-to-goal` rule which now forbids any flying of planes whatsoever. The explanation for this is in our modified `fly` operator:

```
#operator fly(aircraft, city-from, city-to, flevel1, flevel2)
 :at t
 :precond   [t] at(aircraft, city-from) &
            [t] fuel-level(aircraft, flevel1) &
            [t] next(flevel2, flevel1)
 :duration  180
 :context
   :effects [+1] at(aircraft, city-from) := false,
            [+180] at(aircraft, city-to) := true,
            [+1] fuel-level(aircraft, flevel1) := false,
            [+180] fuel-level(aircraft, flevel2) := true
```

After takeoff, the plane is not at `city1` and does not arrive at `city2` until 180 steps later. The definition of `planes-always-fly-to-goal` states that if the plane leaves a city at time t, it should be at a meaningful destination at t + 1.

```
#control :name "planes-always-fly-to-goal"
     forall t, aircraft, city [
          [t] at(aircraft, city) ->
          ([t+1] at(aircraft, city)) |
          exists city2 [
               city2 != city &
               [t+1] at(aircraft, city2) &
               [t] check-if-good-destination(aircraft, city2) ] ]
```

Here, the italicized *`check-if-good-destination(aircraft, city2)`* is a pseudo predicate that represents some test to find out if it is reasonable for the aircraft to visit `city2`. The complete definitions are available in appendix B.2.

At t + 1, the aircraft is not `at` any city at all. It is in the air, traveling somewhere. The rule evaluates to false, meaning the action was not allowed and the planner backtracks. The obvious fix would be to change the rule into the following:

```
#control :name "planes-always-fly-to-goal"
     forall t, aircraft, city [
          [t] at(aircraft, city) ->
          ([t+1] at(aircraft, city)) |
          exists city2 [
               city2 != city &
               [t+180] at(aircraft, city2) &
               [t] check-if-good-destination(aircraft, city2) ] ]
```

This would mean that the duration of the flight is encoded directly in the control rule instead of only in the operator. If the operator's duration is changed, the rule will cease to function correctly. In addition, if variable duration operators were used, as in the Timed version of the domain, the rule would not make sense at all. Instead of saying, do not be at `city2` 180 time steps from now if it is not a reasonable destination, we would like to use the more natural expression, do not fly to `city2` if it is not reasonable. To accomplish this we add a new feature, `flying-to(aircraft, city)`, which is true while the `aircraft` is in the air, flying to the `city`. The definition is simple:

```
#feature flying-to(aircraft, city) :domain boolean :injective
```

To update it we add the following to the effects of the fly operator:

```
[+1] flying-to(aircraft, city-to) := true,
[+180] flying-to(aircraft, city-to) := false
```

And almost identical additions to the faster zoom operator:

```
[+1] flying-to(aircraft, city-to) := true,
[+100] flying-to(aircraft, city-to) := false
```

It is now possible to change the existential formula in the `planes-always-fly-to-goal` rule above as follows:

```
exists city2 [
     [t+1] flying-to(aircraft, city2) &
     [t] check-if-good-destination(aircraft, city2) ]
```

The same problem arises in `planes-always-deliver-passengers-first` in appendix B.2 and again in the `planes-always-fly-to-goal` rule when it checks if someone has boarded a plane. They are corrected by adding another helping feature, `boarding(person, aircraft)`, with the same pattern of use.

Concurrent planning introduces a lot of troubles and it would be easy to feel content with just generating good sequential plans. But this is not a realistic approach if the planner is ever going to do any real world tasks. Consider an airline company that can only schedule one airplane to fly at a time. It might be safe, but not very efficient. TALplanner tries to find a plan that completes the given task as quickly as possible. When creating concurrent plans, as many actions as possible are squeezed in to make the most use of every time step. Relying on the pruning of stupid moves from the search space, doing several actions will probably bring us closer to the solution than doing only one action. The following are the first steps of the plan that TALplanner now generates:

```
0 : (board person1 plane1 city0) [ 20 ]
20 : (fly plane1 city0 city1 fl4 fl3) [ 180 ]
20 : (zoom plane1 city0 city1 fl4 fl3 fl2) [ 100 ]
```

Flying and zooming `plane1` at the same time should be impossible. The planner does not detect this since both actions are possible at time 20, and the effects of the actions do not contradict each other. Only the final fuel levels differ but at different time points resulting in a fuel level of 2 at time 120 and an increase to fuel level 3 at time 200. We again make use of the `$committed` macro by adding the following to the preconditions of `fly` and `zoom`:

```
!$committed(t+1, at(aircraft, city-from), false)
```

It creates a check that the aircraft has not already committed to leaving the city. When `fly` is used, `[+1] at(aircraft, city-from) := false` is committed, rendering zooming impossible and vice versa.

Finally TALplanner rewards us with a short and correct solution for the problem in Figure 4.3.

```
0 : (board person1 plane1 city0) [ 20 ]
20 : (fly plane1 city0 city1 fl4 fl3) [ 180 ]
200 : (board person3 plane1 city1) [ 20 ]
200 : (debark person1 plane1 city1) [ 30 ]
230 : (fly plane1 city1 city0 fl3 fl2) [ 180 ]
410 : (debark person3 plane1 city0) [ 30 ]
;; Plan length 6, maxtime 440
```

Can it be improved? Remember that the STRIPS version never made use of the `zoom` operator. Simple adding of sums now reveals that fly takes 180 time steps and uses one unit of fuel, zoom takes 100 time steps and uses two units of fuel, and refueling takes 73 time units. 180 + 73 is more than 100 + 2 * 73 and therefore we have the opposite situation – `zoom` is always better than `fly`. Commenting out the unwanted `fly` operator yields the following plan:

```
0 : (board person1 plane1 city0) [ 20 ]
20 : (zoom plane1 city0 city1 fl4 fl3 fl2) [ 100 ]
120 : (board person3 plane1 city1) [ 20 ]
120 : (debark person1 plane1 city1) [ 30 ]
150 : (zoom plane1 city1 city0 fl2 fl1 fl0) [ 100 ]
250 : (debark person3 plane1 city0) [ 30 ]
;; Plan length 6, maxtime 280
```

4.3.4 Timed

The Timed version further complicates the timing of the actions and is the version that most closely resembles the original ZenoTravel domain, developed by Penberthy and Weld [5]. Boarding and disembarking times are constant but problem-specific and defined in the respective problem definition as two new features, `boarding-time` and `debarking-time`. Refueling always fills the plane to its maximum `capacity` but consumes time relative to the amount of fuel received and the `refuel-rate` of the aircraft. Each aircraft also has a `fast-speed` and a `slow-speed` with corresponding `fast-burn` and `slow-burn` fuel consumption. The distances between cities are encoded using a `distance(city1, city2)` feature and when an aircraft uses the `zoom` operator to travel from `city1` to `city2`, it will reach its destination after `distance(city1, city2) / fast-speed(aircraft)` time units and consume `distance(city1, city2) * fast-burn(aircraft)` units of fuel. The same goes for the `fly` operator except `slow-speed` and `slow-burn` are used.

TALplanner would not have any problems handling this added complexity if it was not for one of the contest rules. Durations have to be correctly calculated with a precision of three decimals. The whole architecture and semantics of TALplanner was built on discrete integer time. All durations and time calculations are truncated to integers. A solution that had previously been thought out was implemented in order to comply with the precision requirements. Multiply all durations by a factor of a thousand and, before the final plan is printed to the terminal, bring all the figures back with a division using the same factor. This way the planner can continue to work with integer time but present the solution with adequate accuracy. The scale factor is set at the beginning of the specification with a new statement.

```
#timescale 0.001
```

When durations are multiplied, their range extends into the hundreds of thousands of time steps. This prompted changes in the way TALplanner internally represents states and integers and these are discussed in more detail in chapter 5.4.

Few changes are now needed to transform the SimpleTime domain to the Timed version. Some math is necessary to calculate the durations in each operator and, finally, we find that there is a tradeoff between the `fly` and the `zoom` operators. Depending on the speed and fuel consumption values defined in each problem and the situation where the operator is used, it is sometimes better to use `fly` instead of `zoom`. We introduce a new feature, `fly-better-than-zoom(aircraft, city1, city2)`, in the preconditions of `fly` and its negation in the preconditions of `zoom`. It compares the time spent by `fly` and `zoom` to reach the destination and includes the time spent refueling to make up for the fuel loss since zoom requires more fuel. The calculation is not entirely accurate if it later proves unnecessary to fully compensate the fuel used. The plane or the other aircraft might have enough fuel to complete the goals without further refueling. To partly compensate for this, the last clause in the definition permits flying instead of zooming if that makes it possible to get by with one less `refuel` action.

```
// Fly is better than zoom if:
#define [t] fly-better-than-zoom(aircraft, city1, city2):
     // If it's faster wrt speed and refueling.
     ([t] (10000 / slow-speed(aircraft) +
           10000 * slow-burn(aircraft) / refuel-rate(aircraft)) <
          (10000 / fast-speed(aircraft) +
           10000 * fast-burn(aircraft) / refuel-rate(aircraft))) |
     // If zoom is impossible across this distance.
     ([t] distance(city1, city2) * fast-burn(aircraft) >
      capacity(aircraft)) |
     // If zoom has to refuel but fly doesn't.
     ([t] fuel(aircraft) >=
      distance(city1, city2) * slow-burn(aircraft) &
      fuel(aircraft) <
      distance(city1, city2) * fast-burn(aircraft))
```

4.3.5 Discussion

The ZenoTravel domain is easily solved. There are no risks involved in flying a plane to pick up passengers since all the passengers will always fit in the plane and refueling is possible in any city. It is not really possible to get stuck while looking for the solution. The graph of cities is also fully connected so no route planning is necessary. A final version of ZenoTravel, called Numeric, was available in the contest but is not included in the set of domains that we chose to participate in. It is supposedly more difficult and uses a constraint on the number of passengers that an aircraft can carry. The constraint is only enforced in the `zoom` operator but since the numeric domain does not make use of durational operators, it suffers from the same problem as the STRIPS `zoom` operator. It consumes more fuel, limits the number of passengers but does not deliver any advantages because it is no faster than flying. The real difficulty in the Numeric version comes from the complex metrics that are specified in each problem and measures the quality of a solution. E.g. `minimize(total-time + 3 * total-fuel-used)`. TALplanner currently has no way of handling such instructions.

4.4 Depots

The Depots domain is a combination of two classic planning domains, logistics and blocksworld. Blocksworld was introduced in Chapter 2 and the logistics domain is similar to ZenoTravel in that a number of vehicles move objects to specified destinations.

The complete domain and control rule definitions for the domain are available in appendices B.4, B.5 and B.6.

4.4.1 Description

The world of Depots contains locations, trucks, hoists, crates and pallets. Trucks move crates between locations using the `drive` operator. They can move between any two locations and carry any number of crates at the same time. The hoists are distributed among the locations and load crates on trucks or stack crates on top of each other using the four operators `load`, `unload`, `lift` and `drop`. The crates are never put on the ground but instead in stacks on the available pallets or loaded into trucks.

The goal is always to bring the crates into a certain configuration of stacks and is represented by a list of statements deciding which crates should be on top of each other and which crates should rest on which pallets. The complication comes from the fact that if a crate is not at the top of its stack, it cannot be moved until all the other crates blocking it are moved.

To formalize the stacks the same scheme as in the original blocksworld is used. One feature, `on(crate, surface)`, represents a crate being directly on top of a surface, which can be another crate or a pallet, and one feature, `clear(surface)`, states that nothing is on top of that surface.

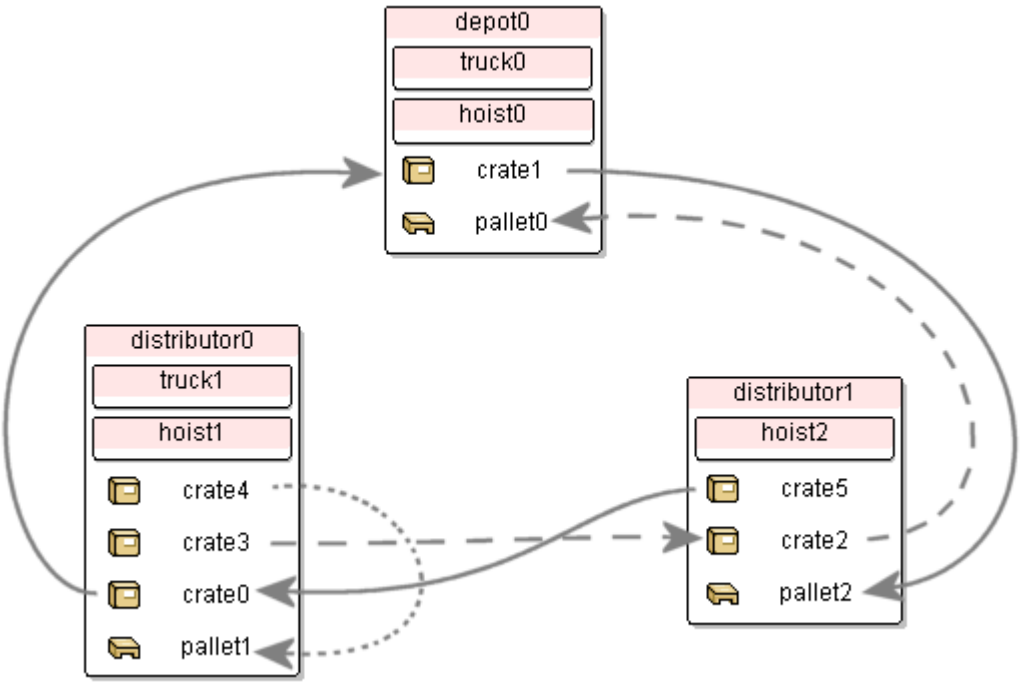

Figure 4.4: Contest problem. A solution rearranges the crates into three stacks: `crate5` on `crate0` on `crate1` on `pallet2`, `crate3` on `crate2` on `pallet0` and `crate4` on `pallet1`.

4.4.2 Control

The blocksworld is a very well-known domain. Good control rules are already available and can be reused here. We used a modified version of Bacchus and Kabanza's rules [9].

Intuitively, we force the planner into only building stacks of crates that are final and which will not be needed to tear down again. Denominating these stacks "good towers", we adopt a recursive definition where a crate is the top of a good tower if the following holds:

1. The crate does not have to be moved to fulfill the goals.
2. If the crate is on another crate, that crate is also a good tower.

In the definitions in appendix B.4, keeping track of which crates are parts of good towers is done in an efficient way by only using the recursive definition to initialize the `goodtower` feature and then updating it directly in the effects of the relevant operators; `lift` and `drop`. The initialization is done in two steps. First define another feature, `goodtower-init`, and then at time 0, use it to initialize the values of `goodtower`.

```
#define [t] goodtower-init(surface1):
     ([t] !need-to-move(surface1)) &
     forall crate, surface2 [
            (surface1 = crate &
             [t] on(crate, surface2)) ->
            [t] goodtower-init(surface2) ]

#dom [0] forall surface [ goodtower(surface) <->
                          goodtower-init(surface) ]
```

The definition depends on another feature, `need-to-move`, which is defined similarly. A crate needs to be moved if

1. it is not on its goal surface or
2. it is on top of another crate that needs to be moved or
3. it occupies a space needed by another crate:

```
#define [t] need-to-move-init(surface1):
     exists crate [
            crate = surface1 &
            (exists surface2 [
                 goal(on(crate, surface2)) &
                 [t] !on(crate, surface2) ] |
             exists crate2 [
                 ([t] on(crate, crate2) &
                  need-to-move-init(crate2)) ] |
             exists surface2 [
                 ([t] on(crate, surface2)) &
                 (exists crate3 [
                      goal(on(crate3, surface2)) &
                      crate3 != crate ]) ] )]

#dom [0] forall surface [ need-to-move(surface) <->
                          need-to-move-init(surface) ]
```

We have yet to make use of the new features in a control rule. All Depots problems can be solved without stacking crates in the wrong order and without ever moving crates that are already stacked in the right order. The defined features do the main part of the work so the rules will be very simple.

```
#control :name "only-create-goodtowers"
      forall t, crate, surface [
          [t] !on(crate, surface) &
          [t+1] on(crate, surface) ->
          [t+1] goodtower(crate) ]

#control :name "only-move-crates-when-necessary"
      forall t, crate, place1 [
          [t] at(crate, place1) ->
          ([t+1] at(crate, place1)) |
          [t] need-to-move(crate) ]
```

Appendix B.4 contains the first rule embedded in the preconditions of the `drop` operator. This avoids unnecessary evaluation of the rule since `drop` is the only operator that creates stacks. The second rule will ensure that good towers are not destroyed. A crate that is part of a good tower cannot be `need-to-move` and therefore must remain in its position in the stack.

It might seem impossible to solve the problems without sometimes stacking blocks in the wrong order. What if one crate needs to be transported to another location but another crate blocks it? Unlike the original blocksworld, we cannot put the top crate temporarily on the table while moving the bottom crate, but instead have to use one of the limited numbers of pallets. If the crate was not supposed to be on that pallet in the goal state, this would violate the `only-create-goodtowers` rule. The trick is to load the top crate into a truck. Trucks can contain any number of crates and the planner will make heavy use of them as storage while building the stacks.

As in ZenoTravel, we would now like to limit the vehicles' movements to only those locations where they can be of any use. Two obviously useful actions are loading crates that are at the wrong location and unloading crates at the right location. But figuring out in which location a crate should be is not as easy as in the ZenoTravel domain where the problem goals simply stated which city a person should end up in. The goals in Depots only specify the order in which the crates should be stacked. A crate's final location depends on the crate beneath it and the crate beneath that crate and so on. At the bottom of the stack there must be a pallet and pallets cannot be moved. This is what finally decides the location of the crate and is suitably formalized as another recursive feature, `need-to-be-at`. Unlike `goodtower` and `need-to-move`, a crate's final location will not change during planning. The `need-to-be-at` feature can be initialized once, before the planning begins, without the need to update it later.

```
#define [t] need-to-be-at-init(crate, place):
     exists pallet [
            goal(on(crate, pallet)) &
            [t] at(pallet, place) ] |
     exists crate2 [
            goal(on(crate, crate2)) &
            [t] need-to-be-at-init(crate2, place) ]

#dom [0] forall crate, place [ need-to-be-at(crate, place) <->
                               need-to-be-at-init(crate, place) ]
```

We then combine the two suggested controls and the new feature to define a rule. Trucks can only move to a location where there is a crate that needs to be at another location or where a crate in the truck needs to be unloaded.

```
#control :name "trucks-always-move-to-goal"
      forall t, truck, place [
          ([t] at(truck, place)) ->
          ([t+1] at(truck, place)) |
          exists place2 [
                 place2 != place &
                 ([t+1] at(truck, place2)) &
                 (([t] exists crate, place3 [
                        generalized-at(crate, place2) &
                        ((need-to-be-at(crate, place3) &
                          place2 != place3)) ] ) |
                  (exists crate [
                    ([t] in(crate, truck) &
                     need-to-be-at(crate, place2)) ])) ]]
```

The rule helps the planner solve the problem in Figure 4.4, but it still has considerable problems doing so.

Table 4.1: The planner's attempt at solving a contest problem.

```
[0,1] Lift(hoist0, crate1, pallet0, depot0)
[0,1] Lift(hoist1, crate4, crate3, distributor0)
[0,1] Lift(hoist2, crate5, crate2, distributor1)
[0,1] Drive(truck0, depot0, distributor0)
[0,1] Drive(truck1, distributor0, depot0)
[1,2] Load(hoist0, crate1, truck1, depot0)
[1,2] Load(hoist1, crate4, truck0, distributor0)
[2,3] Lift(hoist1, crate3, crate0, distributor0)
[2,3] Unload(hoist0, crate1, truck1, depot0)
[2,3] Drive(truck0, distributor0, distributor1)
[3,4] Load(hoist0, crate1, truck1, depot0)
[3,4] Load(hoist2, crate5, truck0, distributor1)
[4,5] Lift(hoist2, crate2, pallet2, distributor1)
[4,5] Unload(hoist0, crate1, truck1, depot0)
[4,5] Drive(truck0, distributor1, distributor0)
[5,6] Load(hoist0, crate1, truck1, depot0)
[5,6] Load(hoist1, crate3, truck0, distributor0)
[6,7] Lift(hoist1, crate0, pallet1, distributor0)
[6,7] Unload(hoist0, crate1, truck1, depot0)
[6,7] Drive(truck0, distributor0, depot0)
[7,8] Load(hoist0, crate1, truck0, depot0)
```

Take a look at the shaded rows in Table 4.1. They appear to describe a wasteful way to load `crate1` onto a truck and suggest improvements in the control rules. That `crate1` was lifted in the first place must mean that it was in the wrong position or otherwise `only-move-crates-when-necessary` would have prohibited that action. Once lifted, loading it into a truck seems like a good idea since the hoist cannot do anything else while lifting `crate1`. Unloading it again without first driving the truck or correcting the stack of crates seem like a dumb idea, and repeatedly unloading it as soon as it has been loaded is plain stupid.

Therefore we define a rule forcing a crate loaded into a truck to stay in the truck until it can be placed on its goal crate or pallet.

```
#control :name "only-unload-crates-when-necessary"
      forall t, crate, truck [
          [t] in(crate, truck) ->
          ([t+1] in(crate, truck)) |
          exists surface, place [
                 goal(on(crate, surface)) &
                 [t] at(surface, place) &
                 [t] at(truck, place) ] ]
```

Combined, these rules solve even the largest contest problems quickly and only minor fine-tuning makes up the difference between them and the rules in the appendix.

4.4.3 SimpleTime

Minimal changes are necessary to comply with the SimpleTime version specification. The `drive` operator has a fixed duration of 10 time steps, `load` 3 steps and `unload` 4 steps. This is easily realized although some caution has to be observed when deciding which effects will take place in the next time step and which are delayed to the end of the operator duration. For example, the `drive` operator sets `at(truck, place1)` to false after one time step and `at(truck, place2)` to true after 10 steps.

As in ZenoTravel's SimpleTime version, we make use of a helper feature that enables the control rules to check vehicles' destinations before they have arrived. This time it is called `driving-to`. Most rules are only interested in checking if a truck stays put or drives off. They are not interested in what the destination is and have no use for the new feature. Only `trucks-always-move-to-goal` has to be updated.

4.4.4 Timed

The Timed version introduces four new features; `distance(place1, place2)`, `speed(truck)`, `weight(crate)` and `power(hoist)`, which are all used when calculating the operator durations. Driving from one place to another has a duration of `distance(place1, place2) / speed(truck)` and loading and unloading crates have durations of `weight(crate) / power(hoist)`.

To provide adequate precision, we again make use of the multiply by a thousand solution and the `timescale` macro. The resulting definitions are available in appendix B.6.

4.4.5 Discussion

The combination of two classic planning domains did not create any new significant difficulties. Control rules from each were readily combined to create good control for the resulting domain.

Again, there was another version of the domain that the TALplanner did not compete in – Depots Numeric version. Load limits have been defined for all trucks, constricting the use of trucks as temporary crate storages and thereby posing a more severe challenge to the planner. The initial testing we have done confirms that the current set of control rules is incapable of solving even smaller problems of the Numeric version.

4.5 DriverLog

DriverLog is yet another logistics domain, this time introducing the concept of truck drivers. A number of packages are transported between locations by trucks and two sets of routes connect the locations. There are links, where trucks travel, and paths, which drivers can walk along when not driving any truck. A truck can only have one driver at a time but can load as many packages as is needed.

The complete domain and control rule definitions for the domain are available in appendices B.7, B.8 and B.9.

4.5.1 Description

In the specification, six operators are defined. Trucks drive between two locations that are connected by a `link` with the help of a driver using the `drive-truck` operator. Drivers walk between locations connected by a `path` with the `walk` operator. Packages are loaded into and unloaded from trucks with `load-truck` and `unload-truck`. Finally, drivers board and disembark trucks with `board-truck` and `debark-truck`.

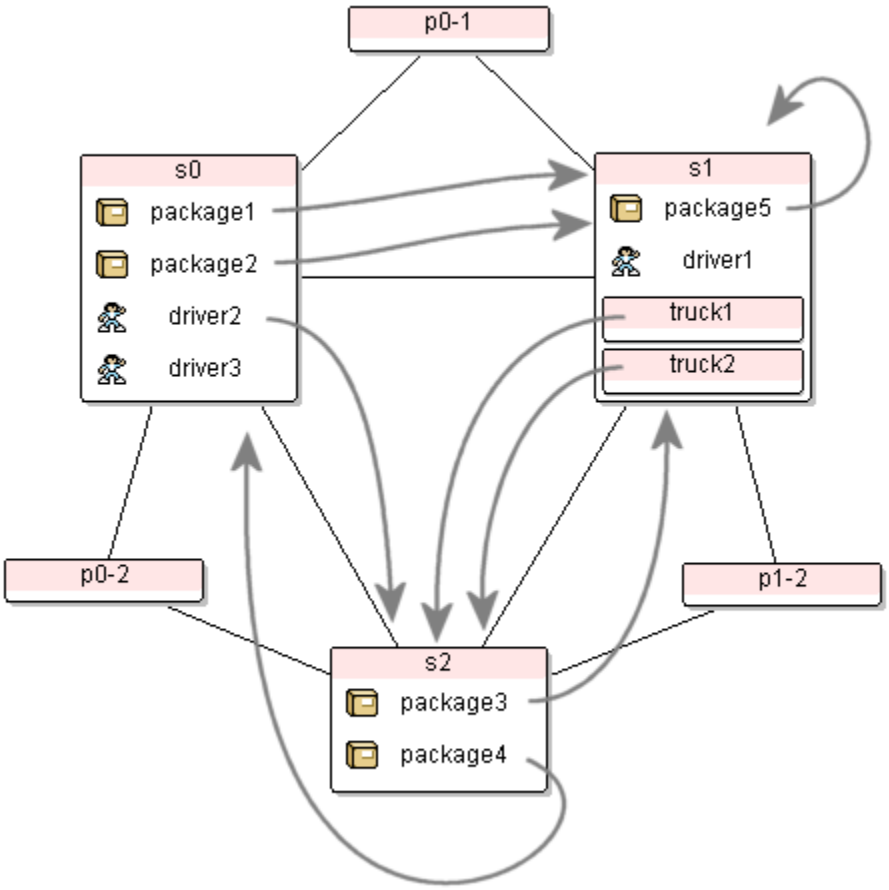

Figure 4.5: Contest problem with the goal locations of the objects pointed out. `S0`, `s1`, and `s2` are proper locations, connected with roads, while `p0-1`, `p0-2`, and `p1-2` are only intermediary nodes, connected with paths.

4.5.2 Control

Using our previously gained experience with logistics domains, we can create and reuse some control rules right from the start. Only load packages into trucks if they actually need to be moved to another location and, if they have been loaded, do not unload them at any other place but their goal destination.

```
#control :name "only-load-when-necessary"
      forall t, obj, location1 [
             ([t] at(obj, location1)) &
             ([t+1] !at(obj, location1)) ->
             goal(!at(obj, location1)) ]

#control :name "only-unload-when-necessary"
      forall t, obj, truck [
             [t] in(obj, truck) &
             ([t+1] !in(obj, truck)) ->
             exists location [
                 [t] at(truck, location) &
                 goal(at(obj, location)) ] ]
```

The next step in both ZenoTravel and Depots was to limit the vehicles' movements to places where they could perform some work by picking up or delivering people or crates. In DriverLog we would like to write two such rules; one controlling the truck movements and one controlling the driver movements. The previous solutions were relatively straightforward: define a feature that evaluates the usefulness of a location given the current state of the world and only allow vehicles to go to locations that pass this test. But this approach does not directly transfer to the DriverLog domain. Since trucks and drivers only travel along certain routes, they may have to pass through one or several intermediary locations before reaching the destination. Looking at the contest problems verifies that this is indeed the case. Although most locations are directly connected by links, not all of them are, and paths never connect two locations but always go through an extra path node.

Our solution is a feature that given a truck or a driver and its current location returns the distance to the closest location out of all locations that pass a test. The test will be the usefulness feature and in the STRIPS version the distance is the number of links or paths that separate the two locations. We can then make trucks and drivers choose only destinations that decrease the value of this new feature. These must be locations that either are, or lies on the way to a location that passed the test.

In the case of the trucks, we define the feature as follows:

```
#define [t] driving-distance-to-reasonable-destination(truck,
                                                       location):
   value(t, $mmin(<to>,
                  [t] reasonable-truck-location(truck, to),
                  driving-distance-between(location, to)))
```

What `$mmin` does is to iterate over all possible instantiations of `to`, which has been defined as a location variable, and for those destinations that are reasonable for the truck to go to, calculate the distances and return the smallest one.

The `reasonable-truck-location` test will be defined later but let us start by looking at `driving-distance-between`. The locations and the links between them form a graph so that finding the shortest distance between two locations amounts to finding the shortest path in a graph. A recursive search would solve the problem:

```
#define [t] driving-distance-between(from, to):
    $ite(from = to,
         0,
         value(t, 1 + $mmin(<intermediate>,
                            [t] link(from, intermediate),
                            driving-distance-between(intermediate,
                                                     to)))))
```

The `$ite` function corresponds to an if-then-else statement where the first argument is the test, the second is the value returned if the test is true, and the third is the value returned if the test is false. In the base case, `from` and `to` are both the same location and the distance must be zero. In the recursive case, we iterate over all locations that are connected to our current location with a link, calculate the distances from these intermediary nodes to the destination and return the shortest distance plus one step to the intermediary node.

This constitutes a depth-first search for the shortest path without any cycle checking. It will quickly bury itself in recursion by going back and forth between nodes or round in circles when the graph contains cycles. Introducing a depth limit is a simple way to prevent this. The longest possible path between locations goes through all nodes in the graph, so this will be our limit.

```
#define [t] number-of-locations:
        value(t, $sum(<location>, true, 1))

#define [t] driving-distance-between(from, to):
  value(t, driving-distance-between-internal(from,
                                             to,
                                             number-of-locations))

#define [t] driving-distance-between-internal(from, to, limit):
    $ite(limit = 0,
         1,
         $ite(from = to,
              0,
              value(t, 1 +
                 $mmin(<intermediate>,
                 [t] link(from, intermediate),
                 driving-distance-between-internal(intermediate,
                                                   to,
                                                   limit - 1)))))
```

The internal version of `driving-distance-between` reduces its extra depth argument each time it calls itself and stops when the limit reaches zero. Returning 1 when the limit is exceeded ensures that the distance a failed search branch finally returns will be greater than any successful path found.

This algorithm works but is very inefficient. Finding the shortest path between two locations in a graph of places and roads seems like a good thing to be able to do, not only for this particular domain, and there are much more effective algorithms than that realized in the formulae above. Therefore such an algorithm was implemented directly in the planner and it is used through two new feature types, `distfeature` and `mindistfeature`, which are described in detail in chapter 5.5. Throwing out all the previous work, we finally arrive at the following definition:

```
#distfeature driving-distance-between(from, to)
             :domain integer :link link

#mindistfeature mindist-driving
                :feature driving-distance-between :domain integer
```

```
#define [t] driving-distance-to-reasonable-destination(truck, to):
     value(t, mindist-driving(
                         location1,
                         to,
                         [t] reasonable-truck-location(truck, to)))
```

These features do the same work much more efficiently and are used in the preconditions of the `drive-truck` operator. Trucks are only allowed to drive from one location to another if the value of `driving-distance-to-reasonable-destination` is reduced.

What remains to be done is defining `reasonable-truck-location`. Ordered as in the definition below, the reasons for a truck to visit a location are:

1. The truck has packages to deliver there.
2. There is a goal that the truck should be there or there is a goal that the driver should be there and no goal preventing him from using the truck to drive there.
3. There are packages to pick up and either the truck is already at the location or no other trucks are already there or on their way there.

```
#define [t] reasonable-truck-location(truck, location):
     exists obj [
            [t] in(obj, truck) &
            goal(at(obj, location)) ] |
     (([t] all-objects-at-their-destinations) &
      (goal(at(truck, location)) |
      (!goal(!at(truck, location)) &
       exists driver [
           [t] driving(driver, truck) &
           goal(at(driver, location)) ]))) |
     (([t] $available(objects-to-move-at(location)) !== 0) &
      (([t] at(truck, location)) |
       !exists truck2 [
            truck2 != truck &
            [t] !empty(truck2) &
            [t] at(truck2, location) ] &
       !exists truck2 [
            truck2 != truck &
            ([t] !empty(truck2)) &
            $committed(t+1, at(truck2, location), true) ]))
```

Two things need further explanation. First, the reason that in the third case, a location is reasonable even if the truck is already there, is that this prevents the truck from driving off before loading the packages. No other location can be closer to the truck so there is no way to reduce the value of `driving-distance-to-reasonable-destination` and the `drive-truck` operator cannot be used. Secondly, `objects-to-move-at(location)` is a resource that keeps track of how many packages are left in the location that needs to be moved to another location. It is initialized once and reduced by one each time a package at the location is loaded into a truck. The initialization is done through the following formula:

```
#dom [0] forall location [
               $init(objects-to-move-at(location)) ==
               $sum(<obj>,
                    [0] at(obj, location) &
                    goal(!at(obj, location)),
                    1) &
               $minimum(objects-to-move-at(location)) == 0 &
               $maximum(objects-to-move-at(location)) == 9999 ]
```

If a feature had been used to model the value, problems would have occurred in the context of concurrent planning. When two trucks load two different packages at the same location simultaneously the feature would be updated twice by an effect such as the following:

```
[+1] objects-to-move-at(location) :=
     value(t, objects-to-move-at(location) - 1)
```

The fact that the effect happens twice does not matter. The value will still be the old value reduced by one instead of the old value reduced by one and then reduced by one again. Fortunately, TALplanner supports true resources [17] that are designed to handle concurrent updates and will give the resource the correct value (see the term resource in Appendix A).

The method we have developed works well with the truck drivers too. All that is needed is a `reasonable-driver-location` feature. If there are packages left to deliver, drivers may walk to trucks that have no driver and use them to deliver the packages. If all packages have been delivered and all the trucks are at their destinations (if they have any), then drivers may walk to their goal destinations. Finally, if all packages have been delivered but some trucks are at the wrong locations, drivers can go to them and drive them to the right locations.

```
#define [t] reasonable-driver-location(driver, location):
     ([t] !all-objects-at-their-destinations) &
     exists truck [
            [t] at(truck, location) &
            ([t] empty(truck)) &
            !$committed(t+1, empty(truck), false) ] |
     [t] all-objects-at-their-destinations &
     ([t] all-nondriven-trucks-at-their-destinations-or-have-
          committed-drivers &
      goal(at(driver, location)) |
      ([t] !all-nondriven-trucks-at-their-destinations-or-have-
          committed-drivers) &
      exists truck [
          [t] at(truck, location) &
          goal(!at(truck, location)) &
          !exists driver2 [
                 driver2 != driver &
                 [t] at(driver2, location) |
                 driving(driver2, truck) ] ])
```

In the final version of DriverLog STRIPS in appendix B.7, a few steps from the plans are shaved off by splitting the `walk` operator into two parts, `walk-choosing-destination` and `walk-towards-destination`. In the first operator the driver chooses one destination to head for and then uses the second operator to walk all the intermediate steps of the path there. The original `walk` operator did not care which location the driver was actually heading for as long as walking reduced the distance to it. With the split `walk` operator it is possible to ensure that two drivers do not choose to walk to the same destination since they both make it explicit which location they are heading towards.

4.5.3 SimpleTime

In the SimpleTime version loading and unloading packages takes 2 time steps, driving a truck 10 and walking 20. Boarding and disembarking remain unchanged. A helper feature `going-to(locatable, location)` was added to represent a driver or truck going to `location` but which has yet to arrive. Both `reasonable-truck-location` and `reasonable-driver-location` are updated to use the feature. The result is available in appendix B.8.

4.5.4 Timed

The Timed version has one interesting change. Two new features, `time-to-walk(location1, location2)` and `time-to-drive(location1, location2)`, specify the duration of walking and driving between any two locations that are connected by a path or a link. The values are different for each problem, which forces our shortest path algorithm to handle weighted edges in the graph. A new attribute in the `distfeature` feature definition specifies what to use as the cost function.

```
#distfeature driving-distance-between(from, to)
             :domain integer :link link :cost time-to-drive

#distfeature walking-distance-between(from, to)
             :domain integer :link path :cost time-to-walk
```

The feature passed with the `cost` attribute must take two arguments of the same type as the `distfeature`.

Remember that in the STRIPS and SimpleTime versions, we only checked that the value of `driving-distance-to-reasonable-destination` was reduced when deciding if a truck was allowed to drive to a location. This does not always work when links have different costs associated with them. A single drive action may reduce the distance using an incredibly costly road link, when it could have used a cheap road link and still reduce it. Instead, we have to make sure that there is no easier way to get closer to a reasonable truck location by taking the cost of the current step into account in the `drive-truck` preconditions:

```
!exists location3 [
    [t] link(location1, location3) &
    [t] driving-distance-to-reasonable-destination(truck,
                                                   location3) +
        driving-distance-between(location1, location3) <
        driving-distance-to-reasonable-destination(truck,
                                                   location2) +
        driving-distance-between(location1, location2) ]
```

4.5.5 Discussion

DriverLog problems set up a graph of locations and travel routes, which makes the choice of which routes to use to deliver packages a hard one. The solution was to use an algorithm to do the work for us. This might seem a bit like cheating but it is not. The contest rules would even allow custom software being developed for each problem domain, at least in the hand-tailored planners' track. Instead, a deadline is set and the domains handed out in advance to all contestants at the same time. This gives some measure of how quickly new domains can be mastered by the planner and thereby how general and flexible it is.

4.6 Rovers

The Rovers domain simulates a simple planetary exploration expedition. A lander vessel carries a number of rovers to the planet surface and provides a communication link back to earth. Each rover has a subset of the general capabilities, retrieving soil samples, retrieving rock samples and capturing images using cameras that support different imaging modes. The cameras are mounted on the rovers, as are storage compartments, one for each rover, which can hold one soil sample or one rock sample. Data from a sample must be sent to the lander by a communication link. All missions revolve around navigating waypoints on the planets surface to collect samples and take images of specified objectives that are only visible from certain waypoints. The terrain may prevent rovers from going directly between two waypoints and different rovers handle different terrain so a list of routes each rover can use is provided.

The complete domain and control rule definitions for the domain are available in appendices B.10, B.11 and B.12.

4.6.1 Description

Nine operators makes Rovers the largest domain we have yet come across in the contest. Two operators, `sample_soil` and `sample_rock`, collect a soil sample or rock sample and pass the `store` to place the sample in as one of the arguments. The effects include setting a feature, `have_soil_analysis(rover, waypoint)` or `have_rock_analysis(rover, waypoint)`, to true, which is then tested in the pre-condition of the `communicate_soil_data` and `communicate_rock_data` operators that sends the data to the lander. Similarly, `take_image` and `communicate_image_data` takes a picture and sends it to the lander, this time including arguments for which camera to use, the imaging mode and which image objective to target. Every time a camera is used, it must first be calibrated on a calibration target objective using `calibrate`. All rover movements are realized through the `navigate` operator that uses the `can_traverse(rover, waypoint1, waypoint2)` feature to check the feasibility of the move. Finally a `drop` action is provided to ready a rover's storage for a new soil or rock sample.

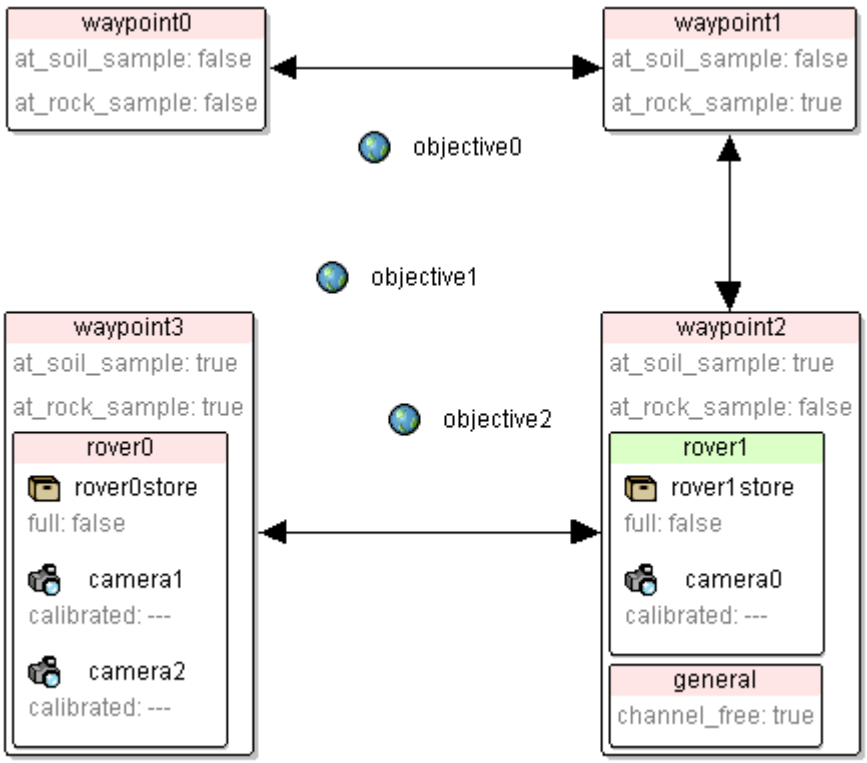

Figure 4.6: Contest problem with arrows showing routes that `rover1` can navigate. The goals for this problem are `communicated_soil_data(waypoint3)`, `communicated_rock_data(waypoint1)` and `communicated_image_data(objective0, high_res)`.

4.6.2 Control

All goals take on the form `communicated_soil_data(waypoint)`, `communicated_rock_data(waypoint)` or `communicated_image_data(objective, mode)` meaning that it does not matter which rover does the sampling and imaging as long as it is done. We therefore create three new features that can be used to prevent a rover from executing an experiment that another rover has already completed.

```
#feature someone_has_rock_analysis(waypoint) :domain boolean
#feature someone_has_soil_analysis(waypoint) :domain boolean
#feature someone_has_image(objective, mode) :domain boolean
```

Following the same control scheme as before, we limit the movements of rovers to locations that seem reasonable by defining a `reasonable-rover-location(rover, waypoint)` feature and use it in a control rule. To achieve the goals, waypoints that have one of the following characteristics have to be visited:

1. A waypoint where some soil or a rock must be sampled.
2. A waypoint that has a clear view of an objective that is to be imaged. Objectives are only visible from waypoints that have the `visible_from(objective, waypoint)` feature set in the problem specification.
3. A waypoint that has a clear view of an objective that is a calibration target for a camera that must be used to capture an image.
4. A waypoint that is visible from the waypoint where the lander is located. Data can only be sent to the lander from waypoints that have the `visible(waypoint1, waypoint2)` feature set in the problem specification.

Collecting all these intuitions into one huge definition yields the following:

```
// A waypoint is reasonable for a rover if:
#define [t] reasonable-rover-location(rover, waypoint):
     // We need to go get a rock sample.
     (goal(communicated_rock_data(waypoint)) &
      [t] at_rock_sample(waypoint) &
      [t] !someone_has_rock_analysis(waypoint) &
      [t] equipped_for_rock_analysis(rover)) |
     // We need to go get a soil sample.
     (goal(communicated_soil_data(waypoint)) &
      [t] at_soil_sample(waypoint) &
      [t] !someone_has_soil_analysis(waypoint) &
      [t] equipped_for_soil_analysis(rover)) |
     // We need to go take an image.
     exists mode, objective [
            goal(communicated_image_data(objective, mode)) &
            [t] visible_from(objective, waypoint) &
            [t] !someone_has_image(objective, mode) &
            ([t] equipped_for_imaging(rover)) &
            exists camera [
                [t] on_board(camera, rover) &
                [t] supports(camera, mode) &
                [t] calibrated(camera, rover) ]] |
     // We need to go calibrate a camera to take an image.
     exists mode, camera, objective [
            goal(communicated_image_data(objective, mode)) &
            [t] !someone_has_image(objective, mode) &
            [t] supports(camera, mode) &
            [t] on_board(camera, rover) &
            [t] !calibrated(camera, rover) &
            [t] calibration_target(camera, objective) &
            [t] visible_from(objective, waypoint) ] |
     // We need to go send rock data to lander.
     exists waypoint2, waypoint3, lander [
            [t] have_rock_analysis(rover, waypoint2) &
            [t] !communicated_rock_data(waypoint2) &
            [t] at_lander(lander, waypoint3) &
            [t] visible(waypoint3, waypoint) ] |
     // We need to go send soil data to lander.
     exists waypoint2, waypoint3, lander [
            [t] have_soil_analysis(rover, waypoint2) &
            [t] !communicated_soil_data(waypoint2) &
            [t] at_lander(lander, waypoint3) &
            [t] visible(waypoint3, waypoint) ] |
     // We need to go send image data to lander.
     exists mode, objective, waypoint2, lander [
            [t] have_image(rover, objective, mode) &
            [t] !communicated_image_data(objective, mode) &
            [t] at_lander(lander, waypoint2) &
            [t] visible(waypoint2, waypoint) ]
```

The effect of the rule is that if a rover navigates to a waypoint, it is guaranteed to be able to do something useful upon arrival. The problem of finding the path from one waypoint to another still remains and although we solved it in the DriverLog domain, the solution cannot be directly applied in the Rovers domain since each rover has its own set of routes between waypoints. Modifying `distfeature` and `mindistfeature` to take an extra argument identifying a rover and applying these to the Rovers domain results in the following definition:

```
#distfeature roving-distance-between(rover, waypoint1, waypoint2)
             :domain integer :link can_traverse

#mindistfeature mindist-roving
                :feature roving-distance-between :domain integer

#define [t] roving-distance-to-reasonable-location(rover,
                                                   waypoint1):
    value(t, mindist-roving(
                rover,
                waypoint1,
                waypoint2,
                [t] reasonable-rover-location(rover, waypoint2)))
```

These tools are used in the same way as in DriverLog. Only allow a rover to navigate somewhere if it decreases the value of `roving-distance-to-reasonable-destination`.

As stated, the rover is now guaranteed to be able to do something useful after navigating, but it is definitely not guaranteed to actually do something useful. Looking at some sample output from the planner as it tries solving the problem in Figure 4.6 it is clear which actions need stricter control. The plan starts by sampling soil at the wrong waypoint, repeatedly calibrating cameras that are not needed and taking pictures of objectives that are not even mentioned in the goal. What we want is to make the plan more efficient by only performing actions that are necessary to fulfill the goals. Only sample soil or rock at waypoints that are specified in the goal, only take pictures that are specified in the goal and only calibrate cameras that can take those pictures.

```
#control :name "only-sample-goal-soil"
      forall t, waypoint [
          [t] !someone_has_soil_analysis(waypoint) ->
          ([t+1] !someone_has_soil_analysis(waypoint)) |
          goal(communicated_soil_data(waypoint)) ]

#control :name "only-sample-goal-rock"
      forall t, waypoint [
          [t] !someone_has_rock_analysis(waypoint) ->
          ([t+1] !someone_has_rock_analysis(waypoint)) |
          goal(communicated_rock_data(waypoint)) ]

#control :name "only-take-goal-images"
      forall t, objective, mode, rover [
          [t] !someone_has_image(objective, mode) ->
          ([t+1] !someone_has_image(objective, mode)) |
          goal(communicated_image_data(objective, mode)) ]
```

```
#control :name "only-calibrate-if-camera-needed"
      forall t, rover, camera [
          [t] !calibrated(camera, rover) ->
          ([t+1] !calibrated(camera, rover)) |
          exists objective, mode [
                 [t] supports(camera, mode) &
                 goal(communicated_image_data(objective, mode)) &
                 [t] !someone_has_image(objective, mode) ] ]
```

Finally, some of the `drop` actions can be skipped. Sampling soil or rock requires free storage space, which is acquired by dropping the contents of the store, but we can delay any dropping until we know that a new sample will actually be collected.

```
#control :name "only-drop-if-neccessary"
      forall t, store [
          [t] full(store) ->
          ([t+1] full(store)) |
          exists rover [
                 ([t] store_of(store, rover)) &
                 exists waypoint [
                     goal(communicated_soil_data(waypoint)) &
                     [t] !someone_has_soil_analysis(waypoint) &
                     [t] at_soil_sample(waypoint) &
                     [t] at(rover, waypoint) &
                     [t] equipped_for_soil_analysis(rover) ] |
                 exists waypoint [
                     goal(communicated_rock_data(waypoint)) &
                     [t] !someone_has_rock_analysis(waypoint) &
                     [t] at_rock_sample(waypoint) &
                     [t] at(rover, waypoint) &
                     [t] equipped_for_rock_analysis(rover) ] ] ]
```

The original contest domain specification forces communication with the surface lander to be serialized. Two rovers cannot send data at the same time. A feature, `channel-free(lander)`, is supposed to take on the value `false` when a rover uses the channel and `true` when the data has been sent. However, there is no way to make this work with TALplanner since sending data only takes one time step. TALplanner does not allow effects to take place at the same time that the action is performed. Setting `channel-free` to `false` must be done one time step later, at the same time that the channel will be free to use again by some other rover. Instead, a resource is defined:

```
#resource sem_communicate_data(lander)
          :domain integer :preference :none

#dom [0] forall lander [
          $init(sem_communicate_data(lander)) == 1 &
          $minimum(sem_communicate_data(lander)) == 0 &
          $maximum(sem_communicate_data(lander)) == 1 ]
```

The resource is initialized to one. All send actions then borrow one unit of the resource during the sending, making other send actions impossible to perform until that one unit has been returned.

```
:resources [+1] :borrow sem_communicate_data(lander) :amount 1
```

4.6.3 SimpleTime

The SimpleTime version changes the durations of all operators except `drop`. Sampling soil takes 10 time steps, sampling rock 8 and sending the data 10. Calibrating a camera takes 5 time steps, taking a picture 7 and sending image data 15. Finally, a rover navigates between two waypoints in 5 time steps.

Only one helper feature, `calibrating(camera)`, was added in order to indicate that the camera has begun calibrating but not yet finished. The `someone_has_rock_analysis`, `someone_has_soil_analysis` and `someone_has_image` features can be used in a similar way by setting them to true before the action is actually completed. It does not matter to the rules that use them if some rover has only started to perform the action or already completed it, e.g. there is no point in a rover navigating to a waypoint to collect a soil sample if another rover is already there and in the process of collecting the soil sample.

As with the lander's communication, a resource, `sem_rover(rover)`, is used to make sure that the rover does not do several things simultaneously. Unlike the lander communication, some of the actions are allowed at the same time. Pictures may be taken while, at the same time, the rover is sampling soil or rock. These actions borrow `sem_rover` non-exclusively while the `navigate` operator borrows the resource exclusively since the rover is never allowed to drive off during another action.

The final definitions are available in appendix B.11.

4.6.4 Timed

The Timed version introduces the interesting concept of energy. Each rover has a limited amount of energy and each action it does consumes some of the energy. The rovers have been equipped with solar panels that recharge the energy but only some of the waypoints that a rover can go to are directly exposed to the sun, which is a requirement for the solar panels to work. This means that a rover can get stuck in the shade, unable to do anything or go anywhere, if it uses its energy unwisely. If that rover carried a camera critical to the mission or could navigate to a waypoint no other rover could get to or if all rovers run out of energy, the goals may be impossible to achieve and the planner will have to backtrack. In addition, the `reasonable-rover-location` definition is now too strict since it does not allow a rover to go to a waypoint just to recharge. It may thus be impossible to find a solution to the problem. Either we can relax the rules and let the planner backtrack and search for a better plan, or we can introduce even stricter rules that keep energy levels in mind when deciding what a rover is allowed to do. The latter approach is taken below.

A rover's energy level is represented by a resource, `renergy(rover)`, which is initialized by an `energy(rover)` feature from the problem instance. The energy is reduced when an action is performed. Navigating is the most costly action and uses 8 units of energy while the other actions use between 1 and 7 units. A new operator, `recharge`, fully replenishes the energy to the maximum of 80 energy units if the waypoint where the rover is located has the `in_sun(waypoint)` feature set. The duration of the `recharge` operator depends on how low the energy level was. Again, the `timescale` statement proves useful, after multiplying all durations with a thousand, to bring the numbers back with appropriate accuracy.

The critical point is when a rover does not have enough energy to reach a waypoint in the sun and recharge. Deciding when this is about to happen requires a new feature that can tell the distance to the closest waypoint having the `in_sun` property.

```
#define [t] roving-distance-to-recharge(rover, waypoint1):
   value(t, mindist-roving(rover,
                           waypoint1,
                           waypoint2,
                           [t] in_sun(waypoint2)))
```

Checking if a rover can afford to perform a certain action is now possible in a function that takes the energy cost of the action as an argument.

```
#define [t] have-enough-energy(rover, fixedpoint):
   exists waypoint [
      [t] at(rover, waypoint) &
      [t] $cast(
             integer,
             fixedpoint,
             value(t,
                   roving-distance-to-recharge(rover, waypoint)))
          * 8.0 <
          ($available(renergy(rover)) - fixedpoint) ]
```

Casting the value of `roving-distance-to-recharge` to a fixed-point value with four decimals is necessary to reduce rounding errors. Multiplying the distance with 8 results in the energy cost to navigate to a recharge waypoint since each navigation step consumes 8 energy units. If the total cost is less than the energy available after performing the action, there is no risk of running out of energy. This test is present in the preconditions of all operators that use energy except navigate.

The `reasonable-rover-location` feature that controls rover movements must be altered to include recharge locations when energy is too low to engage in any other useful activity. In addition to all waypoints that were previously allowed, it is also reasonable for a rover to go to a waypoint, `to`, if that waypoint is exposed to the sun and either the rover does not have enough energy to perform an action and then go recharge, or there do not exist any other waypoints that are both affordable and reasonable to visit.

```
#define [t] reasonable-rover-location(rover, to):
   exists from [
          [t] at(rover, from) &
          (([t] in_sun(to) &
            (([t] $available(renergy(rover)) <
                  $cast(integer,
                        fixedpoint,
                        value(t, roving-distance-to-
                                 recharge(rover, from)))
                  * 8.0 + 8.0) |
             !exists waypoint3 [
                    waypoint3 != to &
                    [t] enough-energy-for-
                        expedition(rover, from, waypoint3) &
                    [t] reasonable-rover-location-
                        dont-care-energy(rover, waypoint3) ])) |
           ([t] enough-energy-for-expedition(rover, from, to) &
            [t] reasonable-rover-location-
                dont-care-energy(rover, to))) ]
```

The STRIPS and SimpleTime definition of `reasonable-rover-location` is now contained in `reasonable-rover-location-dont-care-energy` but `enough-energy-for-expedition` is a new feature that checks if a rover has enough energy to go from waypoint `from` to waypoint `to`, do any one action and still have energy left to get to a recharge waypoint, and is defined as follows:

```
#define [t] enough-energy-for-expedition(rover, from, to):
   [t] $cast(integer,
             fixedpoint,
             value(t, roving-distance-between(rover, from, to) +
                      roving-distance-to-recharge(rover, to)))
             * 8.0 + 8.0 <
             $available(renergy(rover))
```

Again, the multiplication with 8 reflects the cost of navigating and adding 8 to the result ensures that any action can be performed since the actions require between 1 and 8 energy units.

4.6.5 Discussion

Once again the `distfeature` and `mindistfeature` macros are put to great use, even though the Rovers domain is not a typical logistics problem, suggesting that implementing the shortest path algorithm directly in the planner was a good idea.

The energy concept, introduced in the Timed version, significantly increased the level of difficulty for the domain and raised the possibility of reaching a dead end when searching for a solution plan. Our solution, to avoid getting into such situations by always keeping an eye on the rovers' available energy levels and the distance to the nearest recharge locations, is not complete. E.g., a problem with only one rover that has enough energy to fulfill all goals but not enough to go to a recharge waypoint would not be possible to solve with the rules above since they would stop the rover from doing anything but try reaching a recharge location. Even so, the chosen solution has several advantages over the second approach, to loosen the restrictions and let the planner use backtracking to find a correct plan. In general, fewer states have to be examined leading to shorter execution time for the planner and the performance is more consistent over a whole set of problems since each action the planner adds to the plan constitutes steady progress towards achieving the problem goals. In the choice between allowing more search and possibly optimal plans, and using stricter control for a more efficient but incomplete search, we chose the latter.

4.7 Satellite

In the Satellite domain a number of satellites orbit the Earth, each equipped with scientific imaging instruments. The satellites turn in space, targeting stars, planets and interesting phenomena to capture images of them using the instruments different operation modes. These modes can include regular or infrared imaging and spectrographic or thermograph readings but are different for each problem. The planner's task is to schedule a series of observations so that the satellites are used efficiently.

The complete domain and control rule definitions for the domain are available in appendices B.13, B.14 and B.15.

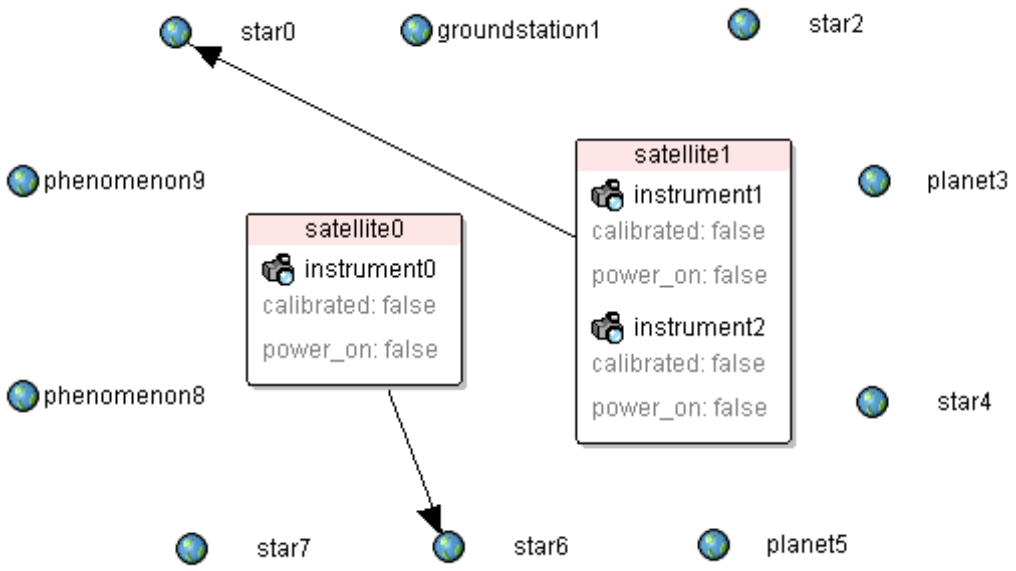

Figure 4.7: Contest problem with arrows showing the directions in which the satellites are initially pointing.

4.7.1 Description

Stars and planets are represented by a `direction` and the satellites can turn between any two directions using the `turn_to` operator. Instruments first need to be activated using `switch_on`, then calibrated at a calibration target with the `calibrate` operator before they can capture images with `take_image`. Each satellite has only enough power to operate one instrument at a time so switching active instruments is always initiated by the `switch_off` operator to deactivate the first instrument.

4.7.2 Control

Since the tasks consist of collecting a number of images, we begin by restricting the use of `take_image` to images that are mentioned in the goal.

```
#control :name "only-take-pictures-of-goals"
      forall t, direction, mode [
          [t] !have_image(direction, mode) &
          [t+1] have_image(direction, mode) ->
          goal(have_image(direction, mode)) ]
```

The next step is to restrict the directions in which satellites turn to those that may actually help in collecting the images. The task is split into a control rule, `only-point-in-goal-directions`, and a definition of goal directions. A satellite is allowed to turn towards a direction to take a picture, to calibrate an instrument or if a goal specifies that the satellite should point in the direction and there is no more work left to do.

```
#define [t] goal_direction(satellite, direction):
     ([t] take_image_possible(satellite, direction)) |
     exists instrument [
          [t] calibration_target(instrument, direction) &
          [t] on_board(instrument, satellite) &
          [t] !calibrated(instrument) &
          [t] power_on(instrument) ] |
     (goal(pointing(satellite, direction)) &
      [t] all_images_collected)
```

The `take_image_possible` feature not only checks if an image in the direction is to be collected but also that it has not already been taken and that the satellite has the necessary instrumentation ready. If the active instrument is not calibrated, the satellite may have to turn towards another direction and calibrate it first.

```
#define [t] take_image_possible(satellite, direction):
     exists mode [
            goal(have_image(direction, mode)) &
            !$committed(t+1, have_image(direction, mode), true) &
            ([t] !have_image(direction, mode)) &
            exists instrument [
                [t] supports(instrument, mode) &
                [t] on_board(instrument, satellite) &
                [t] power_on(instrument) &
                [t] calibrated(instrument) ]]
```

Both the switch_on and switch_off operators are still not regulated by control rules and the planner quickly takes up the habit of repeatedly flipping the power to different instruments on and off. Once an instrument has been powered on and calibrated, using it as much as possible before switching to another instrument seems reasonable. A usefulness feature, putting a value on the usefulness of a particular instrument, helps decide which instrument to power on first.

```
#define [t] usefulness(instrument):
     value(t, $sum(<mode>,
                [t] supports(instrument, mode) &
                 mode_needed_for_goal(mode),
                1))

#define [t] mode_needed_for_goal(mode):
     exists direction [
            goal(have_image(direction, mode)) &
            [t] !have_image(direction, mode) ]
```

Add one to the usefulness score of an instrument for each imaging mode that it supports and that is needed in some goal. This score is then used in a control rule that chooses a satellite's most useful instrument, if it has any.

```
#control :name "use-the-most-useful-instrument"
     forall t, instrument [
          [t] !power_on(instrument) ->
          ([t+1] !power_on(instrument)) |
          ([t] usefulness(instrument) > 0) &
          !exists satellite, instrument2 [
             [t] usefulness(instrument2) > usefulness(instrument) &
             [t] on_board(instrument, satellite) &
             [t] on_board(instrument2, satellite) ] ]
```

Switching off an instrument is only allowed if it is not needed anymore.

```
#control :name "don't-switch-instrument-off-if-you-don't-have-to"
      forall t, instrument [
          [t] power_on(instrument) ->
          ([t+1] power_on(instrument)) |
          !exists mode [
               [t] supports(instrument, mode) &
               [t] mode_needed_for_goal(mode) ] ]
```

Having run out of more or less obvious improvements, analyzing the planner output may still reveal inefficiencies. The satellites often simultaneously decide to turn to the same direction and take a picture. Adding a rule making sure that no other satellite has committed to a certain direction shortens the plans somewhat.

```
#control :name "don't-all-point-in-same-direction"
      forall t, satellite, direction [
          [t] !pointing(satellite, direction) ->
          ([t+1] !pointing(satellite, direction)) |
          !exists satellite2 [
               $committed(t+1,
                          pointing(satellite2, direction),
                          true) ] ]
```

4.7.3 SimpleTime

The SimpleTime version change the duration of some operators. Turning takes 5 time units, switching an instrument on takes 2 units, calibrating it 5 and taking a picture takes 7 time units. A couple of helper features, `turning_towards`, `calibrating`, `power_on_generalized` and `have_image_generalized` keep track of actions that have begun but not completed. The affected control rules are updated with the new features.

Appendix B.14 contains the final definition of the SimpleTime version.

4.7.4 Timed

The Timed version includes two new features, `calibration_time` and `slew_time`. The time it takes to calibrate an instrument is specified for each problem, as is the `slew_time` feature that represents the time it takes for a satellite to turn between any two directions. Neither of these prompts any significant changes to the SimpleTime definition, which is available in appendix B.15.

4.7.5 Discussion

The Satellites domain does not provide a real challenge as long as the planner is only trying to find a correct plan. Finding a short plan is harder. This is especially true in the Timed version. Our control rules does not care in which order the images are collected. The directions and the `slew_time` between them produce a weighted graph that can be searched for an optimal, or sufficiently short, hamilton path. The path then needs to be split and distributed among the satellites to make use of concurrency. Adding all this as control rules seems a bit like overkill.

After the contest, we discovered that the triangle inequality does not hold when turning a satellite between two directions. It is often possible to shorten the slew time by adding `turn_to` actions to several intermediary directions before turning to the goal direction instead of turning towards it directly. The automatic problem generator that created the problem files randomizes the slew times between every pair of directions and does not check for geometrical consistency that would be present in a real world situation. This can be taken advantage of by using the `mindist` feature to find the sequence of turning actions with the shortest combined slew time. Initial testing shows that this approach yields significantly shorter plans when plan length is measured by the time point at which the goals have been fulfilled.

Another improvement would be to change the last clause in the definition of `goal_direction` to allow satellites to turn towards a direction specified in the goals, not when all images have been collected, but when no images are left to collect since some other satellite may have already committed to taking the last picture.

4.8 UMTranslog-2

The UMTranslog-2 domain is another logistics domain but its size and complexity is incomparable to the previously encountered logistics domains in the contest. It is an extension of the UM Translog domain by Andrews et al [10], which was developed specifically to create a challenge for modern planning systems.

Trucks, trains or aircraft transport packages between locations but they must follow strict movement patterns. A few locations are transportation hubs, some are transportation centers while the rest are ordinary locations. A package is only allowed to move up and down through this hierarchy once and only move between two locations in the same layer once. The longest possible route for a package is thus from an ordinary location to a transportation center to a hub to another hub to a transportation center and finally to another ordinary location.

The domain groups locations into cities, which are then grouped in regions. Trucks travel between any two locations in the same city or by an existing road route between two cities. Trains and planes always use predefined routes between transportation centers and hubs.

A great number of restrictions further complicate movements. Packages must be compatible with the vehicle they are loaded into, the vehicle must have enough free space, not be loaded too heavily and not be wider, longer or higher than the route and destination location accepts. Finally, the locations, vehicles and routes must all be available for use.

The complete domain and control rule definitions for the domain are available in appendix B.16.

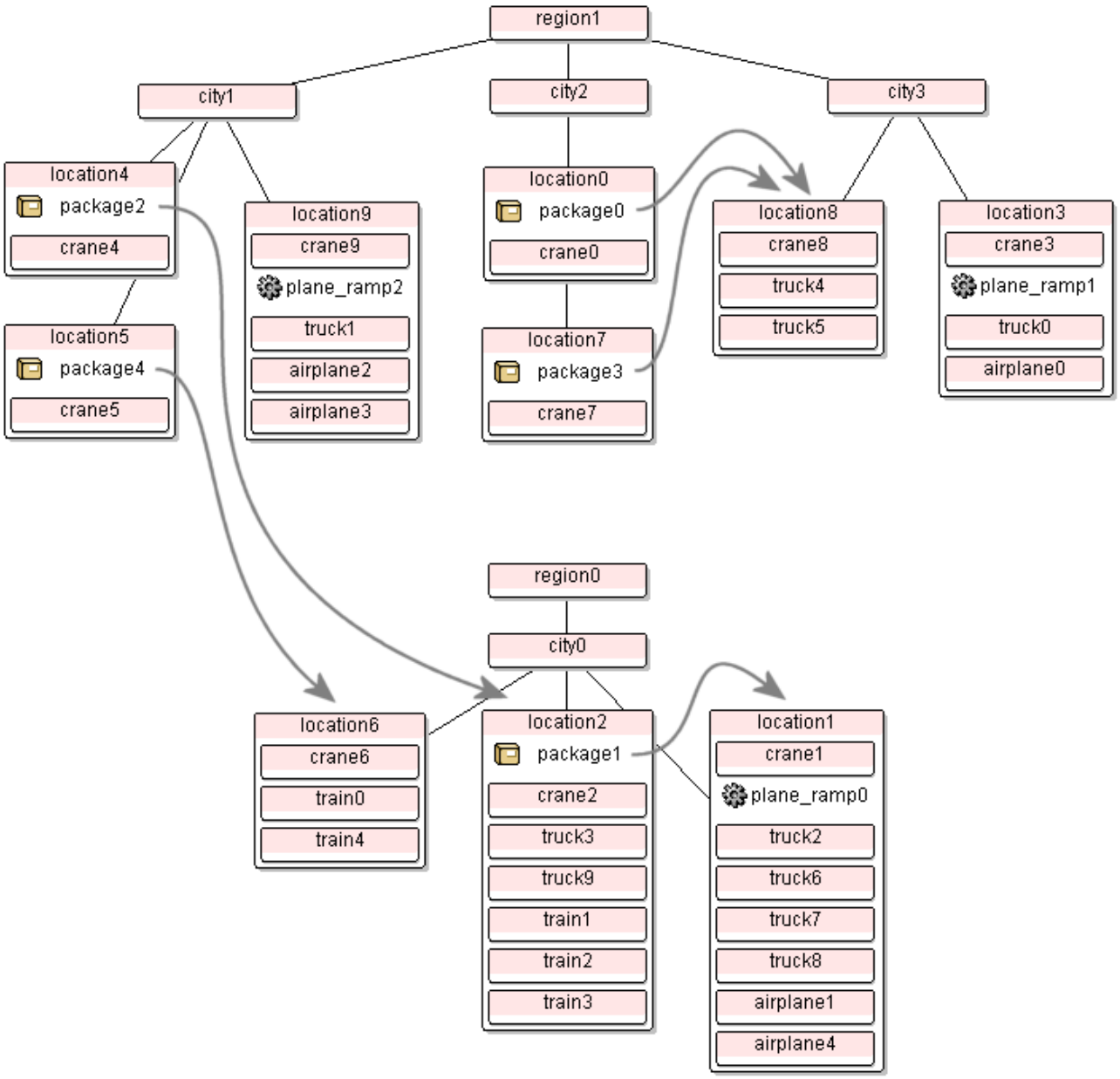

Figure 4.8: Contest problem with arrows showing package destinations.

4.8.1 Description

Many operators have roughly the same purpose and can be grouped together in order to make it easier to get the general idea of the domain. First there are the loading preparation operators, which are different for different types of vehicles. E.g., before loading a tanker truck, the `connect-hose` and `open-valve` operators are used and after the loading is complete, the `close-valve` and `disconnect-hose` operators are required before the truck can go anywhere. Next are the loading operators. They also differ between vehicles and, continuing the tanker truck example, are `fill-tank` and `empty-tank`. The third group is the seven movement operators. Three operators move trucks within cities, one moves trucks along road routes between cities and the remaining three move trains and planes along air and train routes. A set of features keep track of how the packages have been moved between locations to make sure that the transportation pattern described above is used. Finally, there are a number of operators that does not fit into any group. The `collect-fees` operator is used on all packages that need to be delivered at a location with the `deliver` operator. After all packages are delivered, a `clean-domain` operator checks that all vehicle doors and valves are closed and that equipment, like hoses, has been disconnected.

4.8.2 Control

The control rules can also be grouped together and correspond to truck movement, train and plane movement, and package loading and unloading.

As in previous domains, we specify what a reasonable location is and limit vehicle movements to destinations that are reasonable. A truck might want to pick up or deliver a package at the location or, if the truck cannot reach the package's goal location, unload the package at a transportation center to be picked up by another vehicle.

```
#define [t] reasonable-truck-location(vehicle,
                                      location-from,
                                      location-to):
     exists package [
            [t] at-packagel-generalized(package, location-to) &
            [t] !over(package) ] &
     !exists package [
            [t] at-packagev(package, vehicle) ] |
     exists package [
            [t] at-packagev(package, vehicle) &
            goal(delivered(package, location-to)) ] |
     exists package, location-goal [
            [t] at-packagev(package, vehicle) &
            [t] in-wrong-city(package, location-from) &
            [t] in-same-city(location-from, location-to) &
            ([t] tcenter(location-to)) &
            goal(delivered(package, location-goal)) &
            [t] !can-go-by-truck(vehicle,
                                 location-from,
                                 location-goal) ]
```

The `at-packagel-generalized` is a defined feature that helps when referring to a package's location. Unlike `at-packagel`, which is set to false for a package that is lifted by a crane even though the crane is at the same location as the package, the generalized version stays true until the package is loaded into a vehicle.

The definition does not allow trucks to pick up several packages. This makes finding optimal solutions impossible in the general case but simplifies the search for acceptable solutions a great deal. There is an imminent risk that any other packages the truck is carrying will end up at the wrong location if it is allowed to travel about, picking up more packages along the way. Since all packages must move according to the specified pattern of transportation centers and hubs, moving a package that has once arrived at a location that is not a transportation center is not allowed and the package will be stuck there. Restricting trucks to picking up one package at a time avoids this problem.

The `can-go-by-truck` feature is very useful for testing that no fuel, size, weight or road route restrictions are violated.

```
#define [t] can-go-by-truck(vehicle, location-from, location-to):
     [t] $available(rgas-left(vehicle)) >=
         distance(location-from, location-to) * gpm(vehicle) &
     [t] height-cap-l(location-to) >= height-v(vehicle) &
     [t] length-cap-l(location-to) >= length-v(vehicle) &
     ([t] width-cap-l(location-to) >= width-v(vehicle)) &
     (exists city [
            [t] in-city(location-from, city) &
            [t] in-city(location-to, city) ] |
      exists city-from, city-to [
            [t] in-city(location-from, city-from) &
            ([t] in-city(location-to, city-to)) &
            exists route [
                [t] connect-city(route,
                                 road-route,
                                 city-from,
                                 city-to) &
                [t] availabler(route) &
                [t] height-v(vehicle) <= height-cap-r(route) &
                [t] weight-v(vehicle) +
                    $available(rweight-load-v(vehicle)) <=
                    weight-cap-r(route) ] ] )
```

Similar rules are defined for trains and planes and are available in appendix B.16.

The group of loading and unloading restrictions is large but filled with repetitions containing small deviations for different vehicle types. Once again returning to the tanker truck example, we define a rule controlling the opening of tanker valves. For simplicity, the fluids that are transported by tankers are also represented as packages in the domain. Only open a tanker's valve if there is a compatible package at the same location as the tanker that needs to be moved or if the tanker contains a package that needs to be unloaded there.

```
#control :name "only-open-valve-if-needed"
    forall t, vehicle [
       [t] !valve-open(vehicle) &
       ([t+1] valve-open(vehicle)) ->
       exists location [
          ([t] at-vehicle(vehicle, location)) &
          (exists package [
              [t] at-packagel-generalized(package, location) &
              [t] package-vehicle-compatible(package, vehicle) &
                  need-to-move-package-from(package, location) ] |
           exists package [
              [t] at-packagev(package, vehicle) &
              [t] need-to-unload-package-at(package,
                                            location) ]) ] ]
```

An identical rule, with the exception of a negation of the two inner exist statements, control the closing of valves. Two similar rules control the connection of hoses and many more rules control loading and unloading preparations for all other vehicle types.

Additionally, there is a rule, `only-load-packages-into-reasonable-vehicles`, and two definitions of reasonable vehicles, one for trucks and one for trains and planes, which together makes sure that packages are only loaded into vehicles that are actually able to take them to a useful location.

4.8.3 Discussion

UMTranslog-2 is by far the largest domain in the IPC02 contest. 38 operators make the size of the specification intimidating even though there are no SimpleTime or Timed versions. Creating control rules and meeting the contest deadline left no time to get the domain working with concurrent planning. Instead, we had to make do with sequential planning. The situation faced in the contest was not entirely realistic in that no description of the domain was given except the formalization of the operators. Writing control rules is not possible until one has some idea of the intended workings of the domain and therefore time was lost analyzing the functions of peculiar features and operators. Under more typical circumstances, one would start with a description of the problem domain and work out the formalization from it.

Of the 15 contest problems provided, all created automatically by a problem generator, only ten were actually solvable. The remaining five were unsolvable for different, and often obscure, reasons. This could have been intentional, and the ability of a planner to terminate in reasonable time given an unsolvable problem is certainly a valuable quality as many real world problems might be unsolvable, although the impression given was that even the contest organizers were unaware of the fact until after the results from the planners had been collected.

Given more time, the set of control rules provided in appendix B.16 could be improved. They solve the (solvable) contest problems but will fail on other valid problems. If planning speed is less of an issue, more search can be allowed and lower cost plans generated. More and better problems would be needed as guidelines when developing better control rules since the contest problems did not make full use of the intended transportation scheme with transportation centers and hubs.

Chapter 5

Extensions

During the formalization and control rule optimization of the contest domains a number of changes were made to the TALplanner implementation, some of which have been mentioned briefly in Chapter 4. Most of the changes had been planned but not yet implemented due to lack of time, while other changes were prompted by particular difficulties encountered when modeling the new domains in the IPC02 competition. The changes together with two additions, a translation utility and a graphical visualization utility, are described in this chapter.

5.1 Operator Duration

There is no real need to explicitly state an operator's duration since it is implicitly defined in its effects. The effects take place at different time points relative to the actions invocation time and the effect with the longest delay will determine the duration. However, often several effects take place at the end of the duration, which can be a complex expression that will, in the operator definition, have to be duplicated for each of those effects. To save space and make the definition more readable and easier to change, an optional `duration` attribute was added to operator definitions. It accepts a value expression and a variable, which is bound to the value of the expression. This variable can then be used in place of the expression for all effects that take place at the end of the operator duration. A possible usage is displayed below:

```
#operator Drive(truck, city1, city2)
 :at t
 :precond   [t] at(truck, city1)
 :duration  value(t, distance(city1, city2) / speed(truck)) :as t2
 :context
   :effects [+1] at(truck, city1) := false,
            [+t2] at(truck, city2) := true
```

5.2 Prevail Conditions

In the first versions of TALplanner, when all actions spanned exactly one time step, and later, when durative actions were implemented, the need to support conjunctive prevail conditions could be met by introducing additional operator effects. If, for example, a truck must stay put while some cargo is being loaded, a prevailing precondition would be that the truck is `at` a specific `city`. To enforce this, an effect setting `at(truck, city)` to true during the action execution is added. If the truck has moved, that move action would have set `at(truck, city)` to false, creating a conflict that the planner detects.

Adding these complementary effects is an efficient but somewhat limited and unaesthetic way to get around the original problem. Therefore, a `prevail` attribute was added to the operator syntax. It receives, as arguments, a logical formula and a time period during which the formula must hold. These extra conditions are checked in each new state within the given temporal interval.

5.3 Committed Macro

In Chapter 4, the problem of several vehicles trying to solve the same subtask simultaneously was repeatedly encountered and was impossible to solve using regular features in control rules. An action's effects must take place after the invocation time point and it is thus impossible to control what other actions are performed concurrently by evaluating the state at that time. In other words, the current world state will not differ before and after an action is added. Only future states are affected.

The `committed` macro was added as an efficient way of checking a future state without waiting for the planner to reach it. Each time an action is performed, the effects are added to a list of committed facts – facts whose future values are already known. For example if a vehicle, at time t, decides to move from A to B, the fact that the vehicle, at time $t + 1$, must be at B is committed. Another vehicle can then, at time t, decide not to visit B by using the `committed` macro to check if there exists any other vehicles that have already committed to being there at $t + 1$.

The `committed` macro will not be part of future versions of TALplanner but will be replaced by some other construction with a similar role but clearer semantics.

5.4 Decimal Time with Sparse States

As mentioned in Chapter 4, some of the contest domains required operator durations to be calculated with a precision of three decimal digits, which poses a problem for TALplanner since it uses integer time throughout its implementation. To avoid the problem, we multiplied durations by a thousand and later divided them to get the correct figure. This practice introduces another problem. In ZenoTravel for example, the operators have durations of more than a hundred time units. If multiplied, they stretch to a hundred thousand. Originally, TALplanner created an array to contain all the states and allow fast indexing to retrieve any state using a time point. When a single action spans a hundred thousand time points, and a typical plan several million, copious amounts of memory would be consumed. At each of the intermittent time points, some fact about the world state may change as a result of an effect from an operator. However, most of the time points are not interesting and do not correspond to a change in the world state.

This problem was solved by implementing an alternative state representation where, instead of indexing the array with time points, states have an extra field containing the time, and only interesting states, those where some effect takes place, are stored. This means that two consecutive states in the array can represent two completely different time points. All states are still sorted chronologically so if the planner needs to consult a state at a certain time point, a binary search can be performed to find it. The new storage scheme trades some of the access efficiency for reasonable memory use.

5.5 Shortest Path

Several of the contest domains benefited from finding the shortest path between two nodes in a graph as a way to plan vehicles' routes between cities or locations. The well-known Dijkstra algorithm [11] was therefore implemented as a special feature, the `distfeature`. Since roads and paths generally will not change their course during planning, the shortest distances between all nodes in the graph can be pre-calculated before the planning starts, and saved in a sorted list for each node. The list for a node will contain that node's shortest distances to all other nodes in the graph. The `distfeature` is efficiently used in combination with the `mindistfeature`, which returns the distance from a node to the closest node for which some test, passed as an argument in the form of a logical formula, returns true. To implement this the planner need only go through the origin node's sorted distance list, returning the first entry that satisfies the given test.

5.6 Heuristics

Planning is a hard task and even supposing sufficient control rules are supplied, guiding the planner to a solution, the solution actually found may be of inadequate quality (measured in plan length or by some other estimate). Improving the solution is left to the work of the search algorithm, backtracking and exploring vast number of possible plans, trying to find one of higher quality. The planner is still directed by the control rules, but not all domain knowledge is suitably expressed as absolute rules that must never be broken. Some rules are only rules of thumb, correct in nine cases out of ten but failing in special cases. Such rules are called heuristics, and a search algorithm using them, a heuristic search.

A depth-first heuristic search was implemented in TALplanner but was not put to use during the contest. Each domain specifies a set of expressions, estimating the cost to reach the goal. The cost is typically, but not necessarily, the minimum number of time points needed. All the expressions are combined and added to the accumulated cost in the current state, creating an estimate of the total cost of a solution plan that includes the current state. Instead of exploring the states that are possible to reach from the current state in order, the planner chooses the state with the lowest cost first, continuing with the next lowest and so on. When the combined estimate is designed to be admissible, never overestimating the cost to reach the goal, the search constitutes an A* search, which can be proven to be both complete and optimal.

The approach also provides so called anytime planning. After the first solution is found, the planner can continue to search for better solutions. The best plan found so far is remembered and can be returned at any time if the search is aborted.

5.7 Domain Visualization

When trying to find inefficiencies in the planner's performance for a domain, much time is spent analyzing the output and debug information from the planning process. The information includes lists of which predicates are true in each state, and these lists can be both long and difficult to read. To allow the user to tune the output to fit the domain he is working on, a new TALplanner command line parameter was defined, which accepts the name of a visualizer that the user has written. If the parameter is present, instead of printing all state information, the planner makes a call to a method in the named visualizer, passing the current world state in an internal data format as a parameter. The method is then free to print whatever information it sees fits in a more readable format tailored to the current planning domain. Examples of tailoring for a logistics domain might be ordering all information concerning a city into one list or grouping packages together with the trucks they are in.

5.8 Graphical Visualization

Taking domain visualization a step further, a common set of tools that are used to create a customized animated representation of the world state and planning progress for each domain have been developed. This graphical visualization utility is named TPVis. Conceptually the display consists of a set of nodes, which can be container windows or atomic objects, optionally with edges between them. The windows are suitable for displaying cities or vehicles and can contain other windows or objects within the window content area. Things like people and equipment are better represented as atomic objects, which take up less space in the display area. Edges between nodes can indicate any form of relationship between objects, the most obvious interpretation being that two locations are connected by some transportation route.

The graphical visualization animates the actual movements of objects between locations, creating a better instinctive feel for the domain, and the two-dimensional graph display gives an overview that is difficult to provide using only text output. In addition to animating a graph, TPVis simultaneously lists the partial plan leading up to the current state and the problem goals that the planner tries to fulfill.

TPVis also provides a limited form of interactive planning since it, at any point in the planning process, allows the user to force the planner to backtrack and explore a different search branch.

The development of TPVis was not initiated until after the planning contest and no graphical visualization was available during the work on the contest domains.

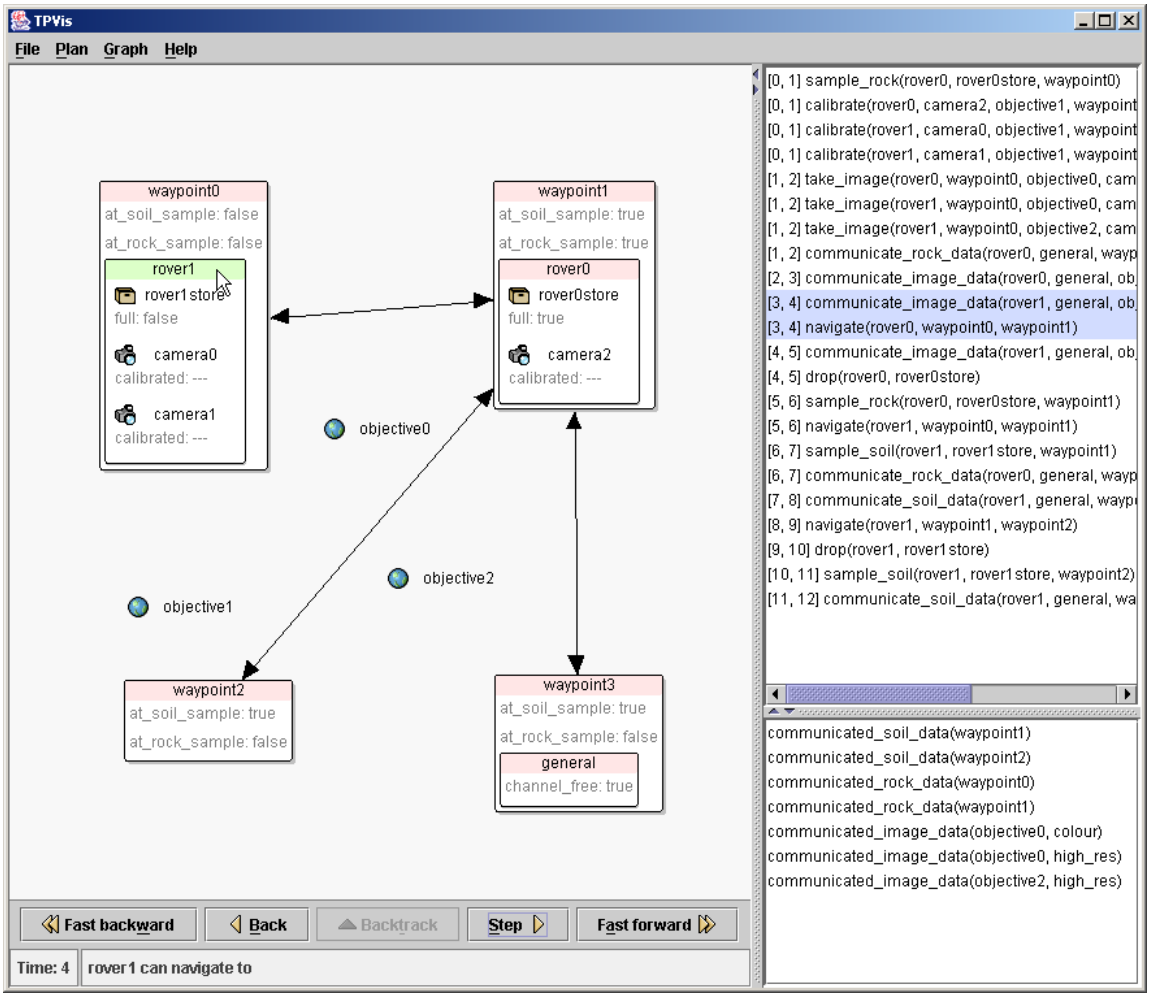

Figure 5.1: Screenshot showing the graphical visualization utility.

5.9 PDDL to TALplanner Translation

Although it is obvious that an automatic translator from PDDL to TALplanner syntax would have been useful, we expected that writing this translator would take more time than translating the competition domains manually and therefore decided to delay the implementation of this utility until after the competition. After spending many hours translating the PDDL definition of the UMTranslog-2 domain, this decision turned out to be questionable. The risk of making an error somewhere in the translation also becomes imminent when dealing with large domains. Such an error might be very difficult to detect if the error relaxes a constraint that is hardly ever used, assuming a finite set of example problems for the domain, and almost impossible to detect without the use of a plan validation utility that uses the corresponding PDDL definition as input. The planner will indeed find a plan, but the plan may violate the constraint that was missed in translation and this fault would have to be detected by inspection.

The automatic translation program was developed after the competition to lessen the amount of work involved in the translation process and reduce the risk of introducing errors in the definition. The program currently does not support the full PDDL syntax but helps a great deal nonetheless.

Chapter 6

Experimental Results

All the competition results have been compiled in one package and are available at the competition web page [3]. This chapter provides comparisons between the results for the three planners that competed in the hand-tailored planning track.

6.1 The Competitors

TALplanner should not need any more introductions, but the other two planners are briefly described below.

6.1.1 TLPlan

TLPlan, developed by Bacchus and Kabanza [9], was one source of inspiration when TALplanner was developed. It is a forward chaining planner that uses first order temporal logic to specify control rules restricting the search. Unlike TALplanner, TLPlan relies on control formula progression. Instead of evaluating the control formula when a new state is created, the formula is progressed using a progression algorithm, making the formula more specific as more actions are added to the plan. If the formula, as a result of the progression, is reduced to *false*, that branch of the search tree is pruned.

6.1.2 SHOP2

SHOP2, developed by Nau et al [13], is a hierarchical task network planner. The planner first formulates the planning problem as a set of tasks that need to be accomplished. The tasks are then repeatedly decomposed into subtasks until primitive tasks, that can be performed using one of the domain operators, are reached. Domain specific knowledge can be used to control the search by specifying complex operator preconditions or by adapting the decomposition of tasks to the domain.

6.2 Machine Specification

All planners used the same machine to generate the plans; a PC supplied by RM plc [12] as a loan system. It is equipped with an AMD Athlon XP-1800+ CPU, 1Gb RAM and the operating system is Mandrake Linux.

6.3 Graphs of Competition Results

Following is a collection of graphs showing the planners' performance. To save space, only the most complex version, the Timed version, of each domain is included. Two graphs are presented for each domain. The first shows a measure of plan cost for the problems and the second shows the time the planners spent generating the solution for the problem. Plan cost correspond to the time point at which the last action in the plan has finished executing and the problem goals have been solved. A lesser value means that the plan accomplishes the goals quicker, for example by making better use of concurrency. The time is measured in milliseconds and a lesser value means that the planner found its solution faster.

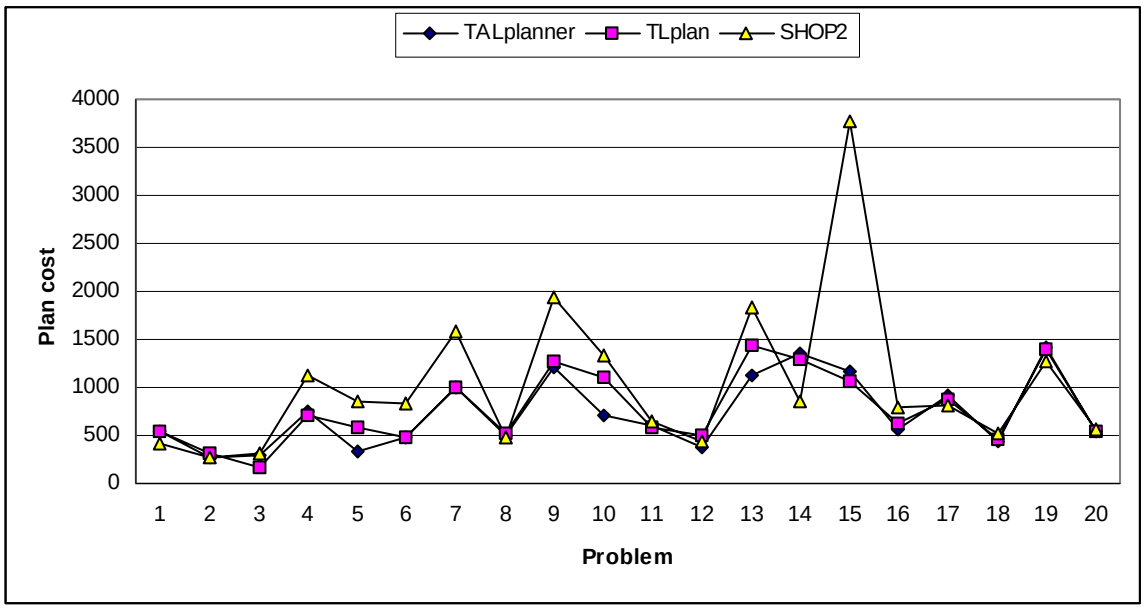

Figure 6.1: Cost graph for ZenoTravel Timed.

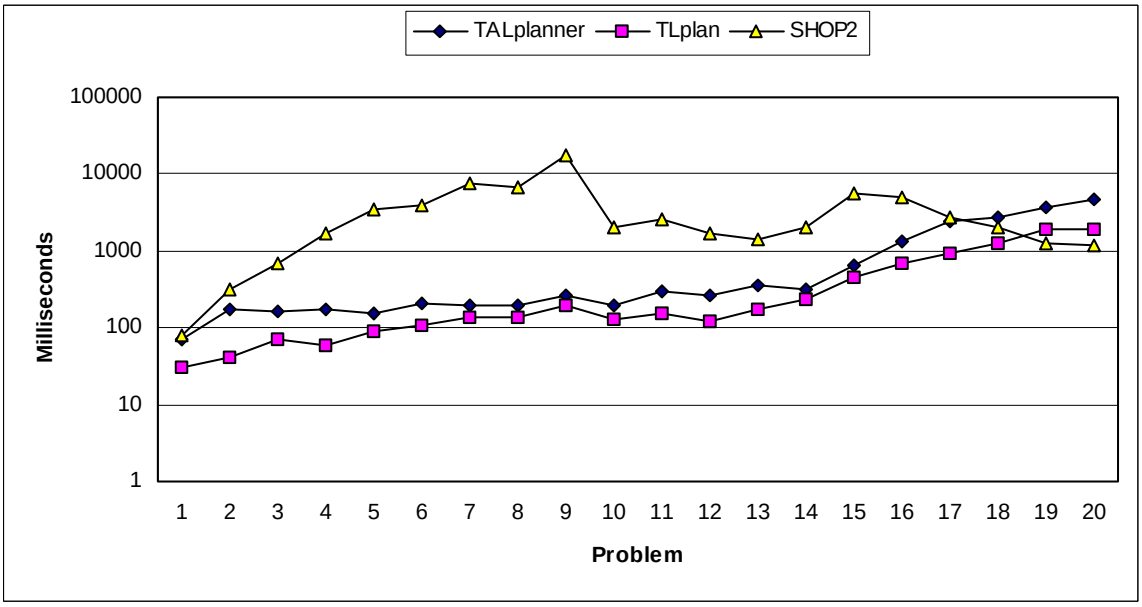

Figure 6.2: Time graph for ZenoTravel Timed.

The graphs in Figure 6.1 and Figure 6.2 show that although TALplanner produced low cost plans, TLPlan matched that cost while solving the problems slightly faster. The difference in architecture between SHOP2 and the other two planners is hinted at by the distinct divergence of its time graph relative to the other planners.

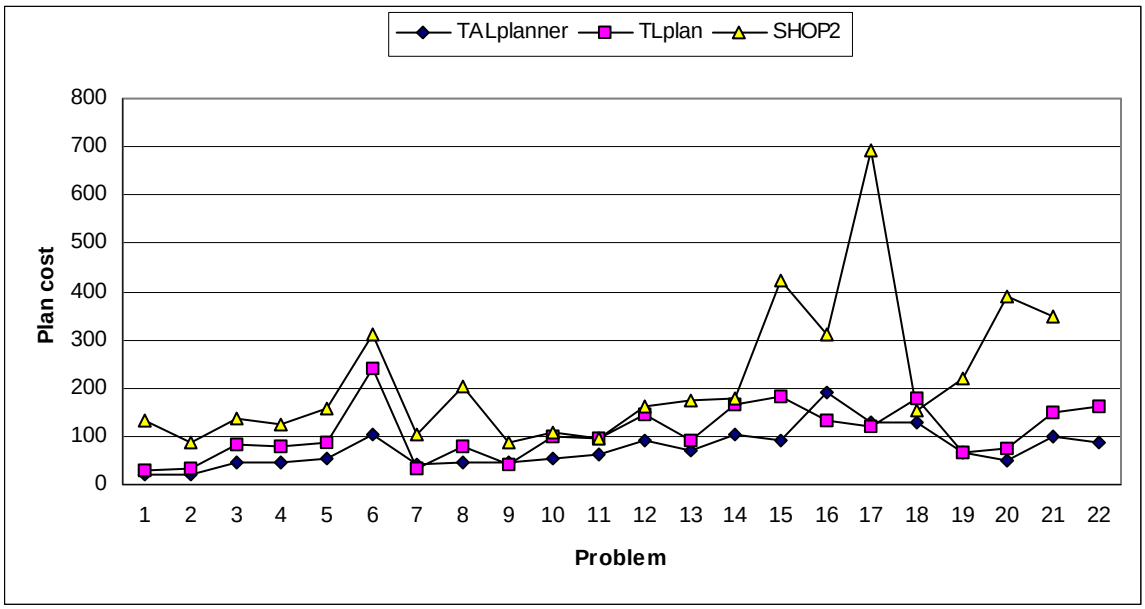

Figure 6.3: Cost graph for Depots Timed.

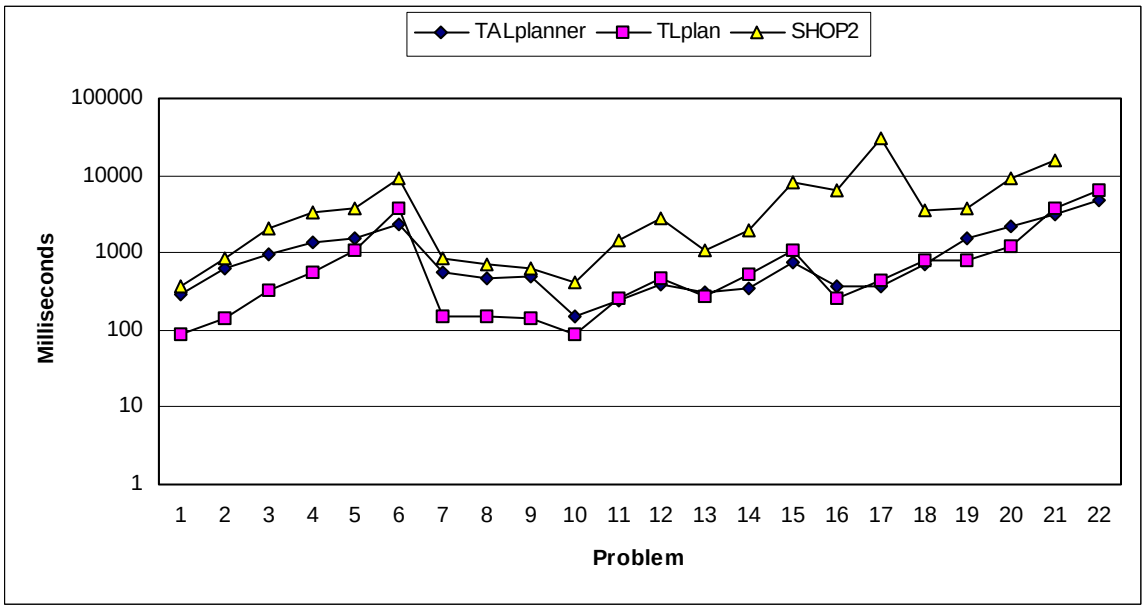

Figure 6.4: Time graph for Depots Timed.

Figure 6.3 and Figure 6.4 show more encouraging results collected from the Depots domain. TALplanner produced plans of significantly lower cost and was sometimes faster than TLPlan.

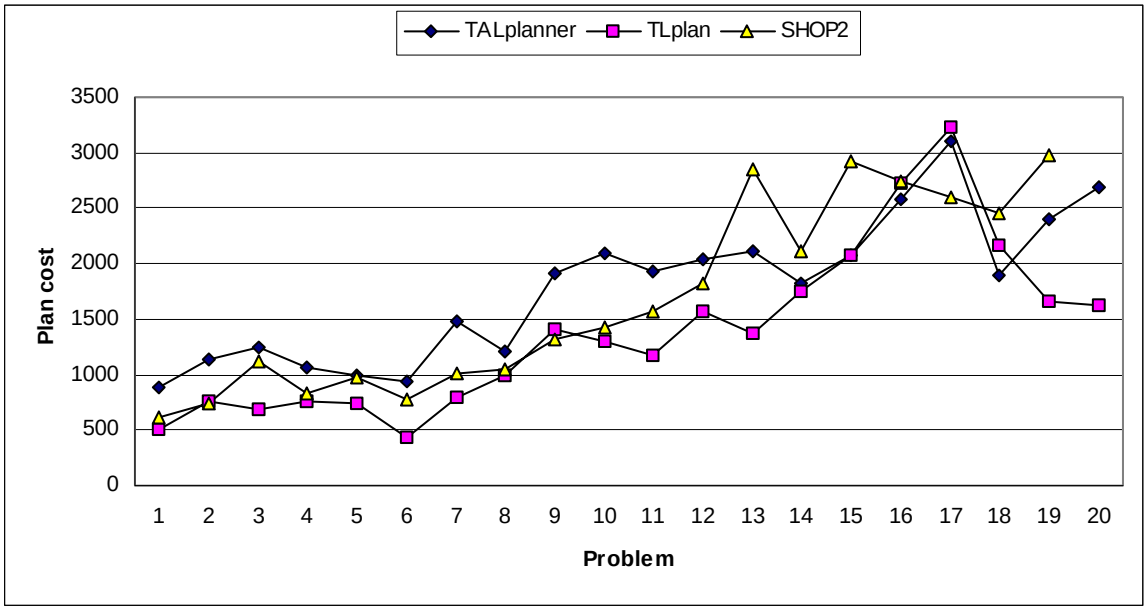

Figure 6.5: Cost graph for DriverLog Timed.

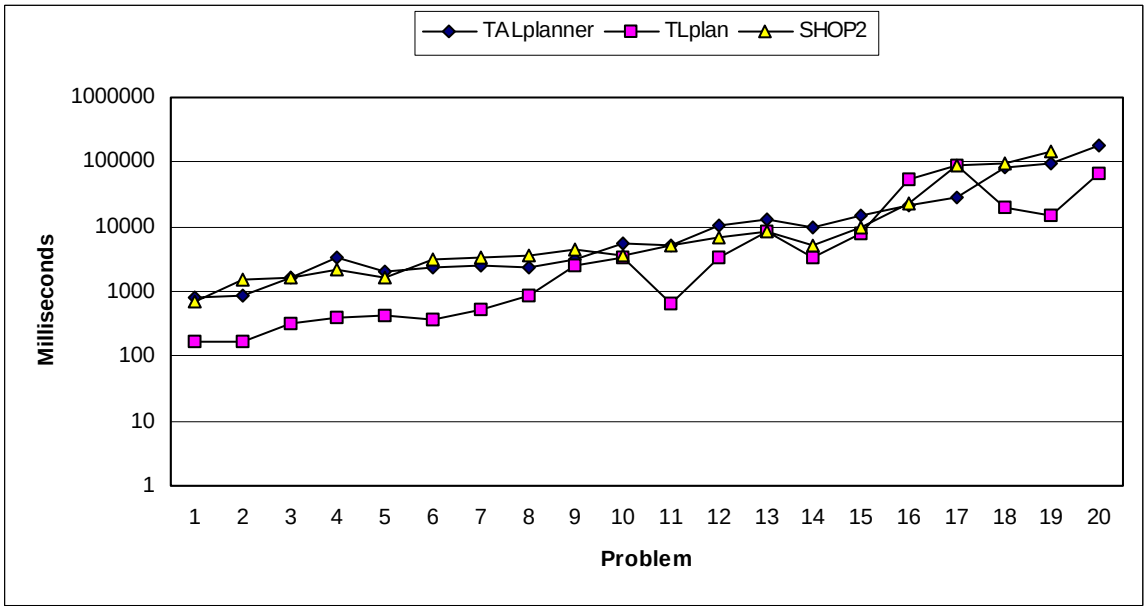

Figure 6.6: Time graph for DriverLog Timed.

In Figure 6.5 and Figure 6.6 TALplanner is in close competition with SHOP2 but TLPlan leaves them both far behind.

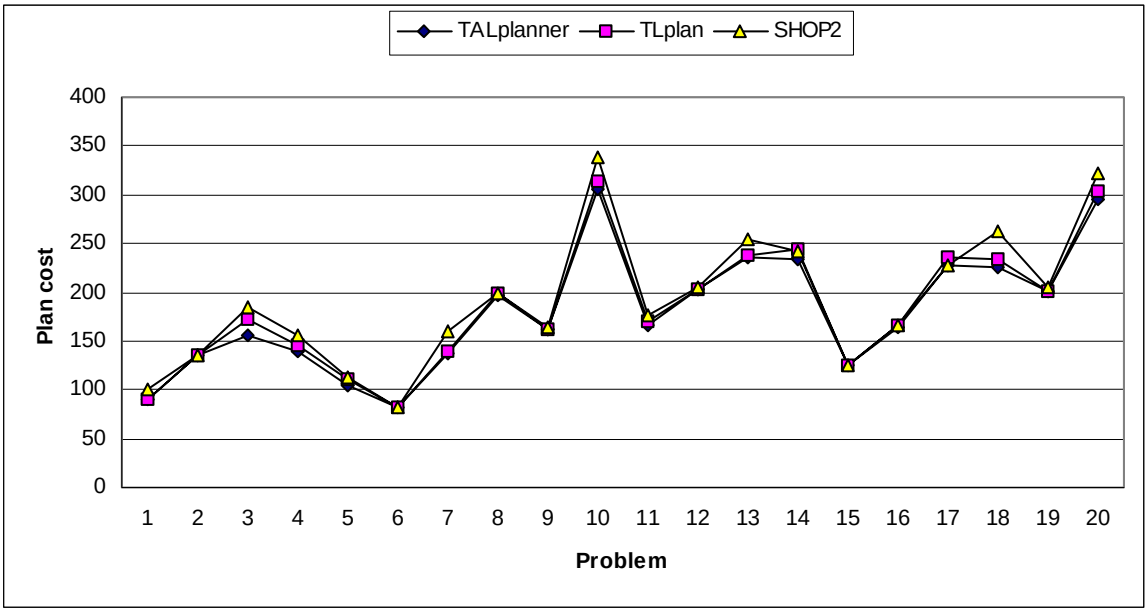

Figure 6.7: Cost graph for Rovers Timed.

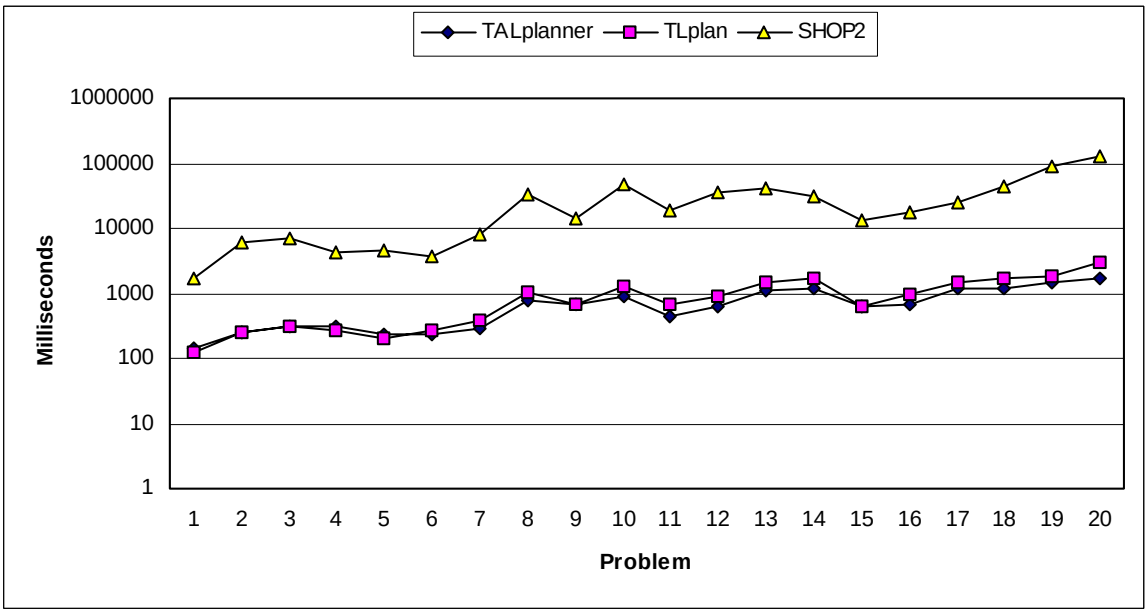

Figure 6.8: Time graph for Rovers Timed.

Figure 6.7 shows a remarkable symmetry between the three planners in the Rovers domain, although TALplanner consistently has slightly lower cost plans. Figure 6.8 clearly indicates TALplanner as the fastest planner in this domain.

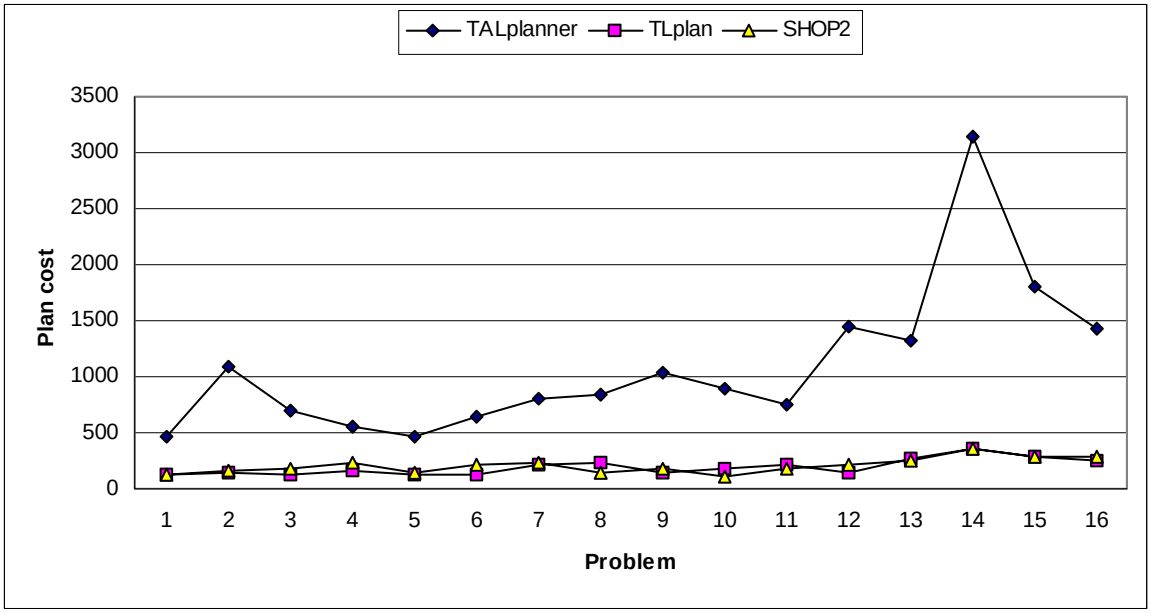

Figure 6.9: Cost graph for Satellite Timed.

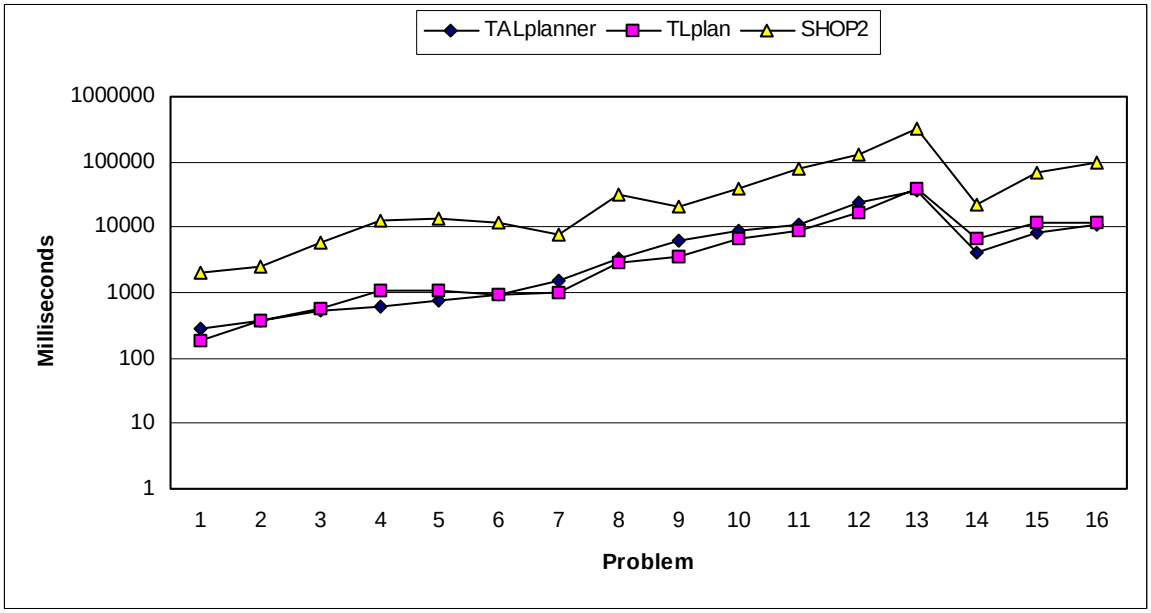

Figure 6.10: Time graph for Satellite Timed.

The graph in Figure 6.9 displays disastrous cost results for the plans by TALplanner in the Satellite domain. One can only guess that taking advantage of the fact that the triangle inequality was not fulfilled when turning the satellites and implementing the suggestions in section 4.7.5 would improve the figures enough to compete with TLPlan and SHOP2. Figure 6.10 does not provide much consolation by showing that the plans were produced quite quickly.

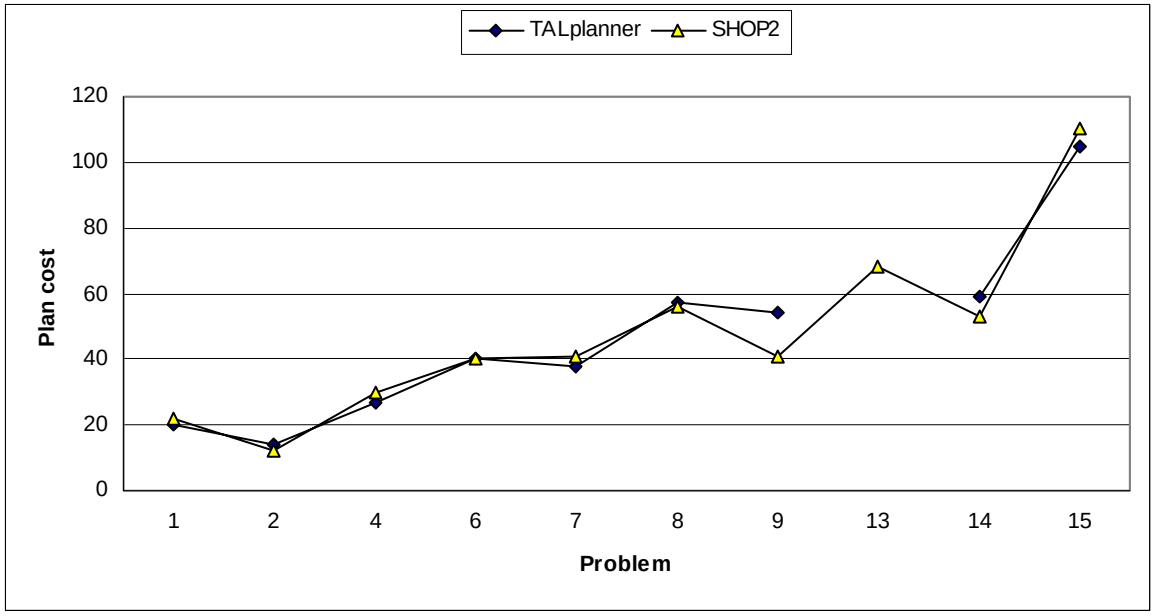

Figure 6.11: Cost graph for UMTranslog-2.

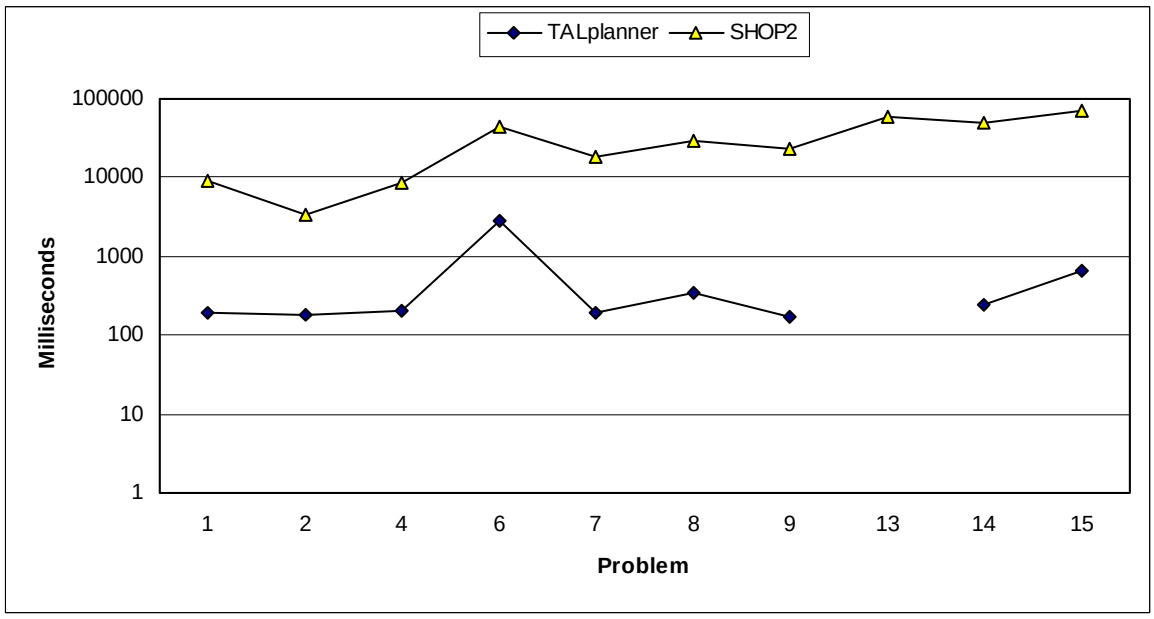

Figure 6.12: Time graph for UMTranslog-2.

TLPlan chose not to participate in the UMTranslog-2 domain, which might have been a wise decision. If the time we put into this domain had been spent developing control rules for the Numeric or Complex versions of the other domains, our problem coverage would most certainly have been better. The gaps in the numbering of the problem files correspond to the unsolvable problems mentioned in Chapter 4 and the gap in TALplanner's graph is the result of an incorrect plan generated for problem 13. Figure 6.11 shows the close race in plan cost between TALplanner and SHOP2 while Figure 6.12 separates the planners through a huge speed difference. TALplanner is almost a hundred times faster in this domain.

6.4 The Prizes

The contest organizers judged the planners' performances based on the coverage, i.e. the number of domains and domain versions each planner entered in the contest, the cost of the solutions and the speed with which they were generated, and the success ratio, i.e. the ratio between attempted problems and solved problems. Coverage was especially important since one of the goals of the competition is to push the capabilities of planning systems forward.

Two prizes were awarded in the hand-tailored competition track based on the above criteria. TLPlan was recognized as demonstrating *distinguished performance of the first order* and SHOP2 as demonstrating *distinguished performance*.

Chapter 7

Conclusions

This thesis has provided an introduction to TALplanner and the International Planning Competition, described in detail the modeling of the competition domains for use by TALplanner, listed the modifications and additions that have been made to the planner, and presented the competition results. This chapter will conclude with a discussion of what has been learned in the process and point to future directions of research in the area of domain dependent planning.

7.1 Discussion

A most interesting observation can be made regarding TLPlan's and TALplanner's use of domain dependent knowledge. Both teams refined the control rules to such a degree that the planners almost never had to backtrack. Instead of searching for the solution, strict control directed the planner straight to it. This does not mean that only one solution can be found. If the user forces the planner to backtrack when the first plan is complete, on the grounds of inadequate quality or for some other reason, the planner will backtrack and generate alternative solutions. It does mean that the planner is guaranteed to find a solution without using any search, acting more like an algorithm than a planner. This raises the question of which real world problems are really suitably solved by planning. Domain independent planners are more often than not too inefficient to handle problems of reasonable size and domain dependent planners behave like an algorithm, so why not use an algorithm instead? The answer may be problems where near optimal solutions are needed but no known algorithm exists which can provide them. In this case, a planner might use a combination of control rules and heuristics to search for such solutions.

Another observation is the problem of overly specific control rules. When the intention is to prevent backtracking, the result is often control rules so restrictive that an optimal solution cannot be achieved. It is too easy to think algorithmically, i.e. how should the planner solve this problem, and force the planner to follow that algorithm instead of thinking declaratively, i.e. what sequences of actions are always disadvantageous in this problem domain. Finding the optimal solution is certainly not always desired and most often intractable anyway, but in those cases where finding it is an objective, extreme care has to be taken not to write any control rule banning it by mistake.

7.2 Future Work.

Heuristic search was implemented but not tested in more than a few of the domains. This situation will have to be corrected by examining the benefits offered by the use of heuristics in more detail. Also, other search strategies than A* search can be implemented and tested, hill climbing being a prime candidate.

The most important development would be a way for the planner to discover control rules by itself. Currently a lot of work needs to be put into the control rules before good quality plans emerge. Any automation of this process would surely save time and be of great value, but research on this by others has already begun and it is not easily done. One approach could be to analyze the domain definition, trying to extract rules from it. Another approach would be to let the planner solve a set of sample problems, extracting rules from patterns found in the search process and the solution plans.

Bibliography

[1] P. Doherty, J. Gustafsson, L. Karlsson and J. Kvarnström, *(TAL) Temporal Action Logics: Language Specification and Tutorial*. Electronic Transactions on Artificial Intelligence, Vol. 2 Issue 3-4: 273-306, 1998.

[2] P. Doherty and J. Kvarnström, *TALplanner: A Temporal Logic Based Planner*. AI Magazine, Fall Issue, 2001.

[3] D. Long and M. Fox, *International Planning Competition 2002* http://www.dur.ac.uk/d.p.long/competition.html, May 2002.

[4] J. Kvarnström, *Applying Domain Analysis Techniques for Domain-Dependent Control in TALplanner*. Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02): 101-110, AAAI Press, 2002.

[5] J. Penberthy and D. Weld, *Temporal Planning with Continuous Change.* Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), Vol. 2: 1010-1015, Seattle, Washington, USA. AAAI Press/MIT Press, 1994.

[6] R. Fikes and N. Nilsson, *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artificial Intelligence Vol. 2 Issue 3: 189-208, 1971.

[7] M. Fox and D. Long, *An Extension to PDDL for Expressing Temporal Planning Domains.* http://www.dur.ac.uk/d.p.long/pddl2.ps.gz, August 2002.

[8] D. McDermott et al, *PDDL – The Planning Domain Definition Language.* ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz, August 2002.

[9] F. Bacchus and F. Kabanza, *Using Temporal Logic to Control Search in a Forward Chaining Planner*. New Directions in AI Planning: 141-153, ISO Press, 1996.

[10] S. Andrews, B. Kettler, K. Erol and J. Hendler, *UM Translog: A Planning Domain for the Development and Benchmarking of Planning Systems.* http://www.cs.umd.edu/projects/plus/UMT/umt.ps, September 2002.

[11] E. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik 1: 269-271, 1959.

[12] RM plc company web page. http://www.rm.com, September 2002.

[13] D. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. *Total-Order Planning with Partially Ordered Subtasks*. Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001): 425-430, Seattle, Washington, USA, 1994.

[14] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, J. Wiklund. *The WITAS Unmanned Aerial Vehicle Project*. Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00): 747-755, ISO Press, 2000.

[15] R. Fikes, P. Hart, N. Nilsson. *Learning and Executing Generalized Robot Plans*. Artificial Intelligence Vol 3, Issue 4: 251-288, 1972.

[16] S. Chien, R. Kambhampati, C. Knoblock, *Fifth International Conference on AI Planning and Scheduling*. http://www-aig.jpl.nasa.gov/public/aips00/aips_home.html, September 2002.

[17] J. Kvarnström, P. Doherty, P. Haslum, *Extending TALplanner with Concurrency and Resources*. Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-00), ISO Press, 2000.

[18] P. Doherty, J.Kvarnström, *TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner*. Proceedings of the 6th International Workshop on the Temporal Representation and Reasoning (TIME99), Orlando, Florida, 1999.

[19] J. Kvarnström, P. Doherty, *TALplanner: A Temporal Logic Based Forward Chaining Planner*. Annals of Mathematics and Artificial Intelligence (AMAI), Volume 30: 119-169, 2001.

Appendix A

Terminology

Domain
A set of related operators, predicates, and data types make up a planning domain.

Problem
A problem is a set of objects, a list of predicates that specify an initial state by enumerating the facts that are true before any planning has begun, and a set of goals that must be fulfilled. The planner receives a domain definition and a problem as input and generates a solution plan (if possible) as output.

Operator
An operator is a formal definition of an action that is possible to perform in a specific planning domain. It consists of a set of preconditions, which must hold for the operator to be applicable, and a set of (possibly conditional) effects, which are realized if the operator is applied. Operators contain variables that represent objects in a domain problem. Often these variables are typed and hence limited to a subset of the objects in the problem, e.g. all vehicles in a logistics problem.

Action
An action is an instantiated operator. All variables in the operator have been instantiated to specific objects in a planning problem.

Plan
In the case of sequential planning where all operators take one time step, a plan is a sequence of actions. If time is explicit in the operators, a plan must include timed actions. When concurrent planning is used, a plan is a set of timed actions, not necessarily in a sequence.

State
A state is a set of assignments of values to state variables and describes the state of the world at a specific time point. Many planners support only boolean state variables, which are either true or false, but state variables in TALplanner are based on TAL and can represent boolean values, objects in the domain, or numeric values. The only limitation is that the value domain must be finite. E.g. if a state variable is of an integer type, a lower and an upper bound must be supplied.

Feature
A TALplanner state variable representing an object, a boolean value, or a numeric value is called a feature.

Resource
Unlike an ordinary feature, a resource, `res`, has multiple values associated with it: the amount that has been borrowed in a state (`$borrowed(res)`), the amount borrowed nonexclusively (`$borrow_nonex(res)`), the amount produced (`$produced(res)`), and the amount that has been consumed (`$consumed(res)`). These values are the sums of the four corresponding types of resource effects that took place in the state. Given these values and the amount that was initially available (`$init(res)`), the planner can automatically calculate how much will be available in the state after all action effects have taken place (`$available(res)`). This value must be between the minimum value allowed (`$minimum(res)`) and the maximum value allowed (`$maximum(res)`). When considering only sequential planning, a resource can be modeled with a regular fluent by assigning new values to it in the effects of an operator although this method does not work with concurrent planning where proper resource handling is required.

Appendix B

Domain Definitions

B.1 ZenoTravel STRIPS

```
#domain thing :elements {}
#domain aircraft :parent thing :elements {}
#domain person :parent thing :elements {}
#domain city :elements {}
#domain flevel :elements {}

#domain integer :integer :lb 0 :ub 10000

#feature at(thing, city) :domain boolean :injective
#feature in(person, aircraft) :domain boolean :injective
#feature fuel-level(aircraft, flevel) :domain boolean :injective
#feature next(flevel, flevel) :domain boolean :double-injective

// An aircraft needs-to-visit a city if:
// 1: It's a goal.
// 2: It carries a passenger going there.
#feature needs-to-visit(aircraft, city) :domain boolean :noinit
#deffeature in-wrong-city(thing) :domain boolean
#deffeature all-persons-at-their-destinations-or-in-planes :domain boolean

// The following assertions enable the planner to optimize the application of
// some control rules.
// No object can be both a person and an aircraft.
#assert forall person, aircraft [ person != aircraft ]
// No person can be both in an aircraft and in a city.
#assert forall t, person, aircraft, city [
               [t] in(person, aircraft) -> !at(person, city) ]



#operator board(person, aircraft, city)
 :at t
 :precond       [t] at(person, city) &
                [t] at(aircraft, city)
 :context
   :effects     [+1] at(person, city) := false,
                [+1] in(person, aircraft) := true,
                [+1] at(aircraft, city) := true // Prevail
 :context
                // Loop through all cities and look for the person's
                // destination. Add that to the places that the aircraft
                // needs to visit.
   :forall      city2
   :precond     goal(at(person, city2))
```

```
   :effects     [+1] needs-to-visit(aircraft, city2) := true

#operator debark(person, aircraft, city)
 :at t
 :precond       [t] in(person, aircraft) &
                [t] at(aircraft, city)
 :context
   :effects     [+1] in(person, aircraft) := false,
                [+1] at(person, city) := true,
                [+1] at(aircraft, city) := true // Prevail
 :context
                // The aircraft has to visit this city again if it's a goal and
                // it first has to travel somewhere else to drop a passenger
                // off.
   :precond     !(goal(at(aircraft, city)) &
                        (exists person2 [
                                $committed(t+1, in(person2, aircraft), true) &
                                goal(!at(person2, city)) ] |
                         exists person2 [
                                [t] in(person2, aircraft) &
                                goal(!at(person2, city)) ]))
   :effects     [+1] needs-to-visit(aircraft, city) := false

#operator fly(aircraft, city1, city2, flevel1, flevel2)
 :at t
 :precond       [t] at(aircraft, city1) &
                [t] fuel-level(aircraft, flevel1) &
                ([t] next(flevel2, flevel1)) &
                // Should be generated automatically:
                !$committed(t+1, at(aircraft, city1), false)
 :context
   :effects     [+1] at(aircraft, city1) := false,
                [+1] at(aircraft, city2) := true,
                [+1] fuel-level(aircraft, flevel1) := false,
                [+1] fuel-level(aircraft, flevel2) := true

//#operator zoom(aircraft, city1, city2, flevel1, flevel2, flevel3)
// :at t
// :precond     [t] at(aircraft, city1) &
//              [t] fuel-level(aircraft, flevel1) &
//              [t] next(flevel2, flevel1) &
//              [t] next(flevel3, flevel2)
// :context
//   :effects   [+1] at(aircraft, city1) := false,
//              [+1] at(aircraft, city2) := true,
//              [+1] fuel-level(aircraft, flevel1) := false,
//              [+1] fuel-level(aircraft, flevel3) := true

#operator refuel(aircraft, city, flevel, flevel1)
 :at t
 :precond       [t] fuel-level(aircraft, flevel) &
                [t] next(flevel, flevel1) &
                ([t] at(aircraft, city))  &
                [t] $index(flevel) == 0 // Only refuel when empty.
 :context
   :effects     [+1] fuel-level(aircraft, flevel) := false,
                [+1] fuel-level(aircraft, flevel1) := true,
                [+1] at(aircraft, city) := true // Prevail



// Initialize the needs-to-visit defined feature, which is later updated in the
// operator definitions.
#dom forall aircraft, city [
```

```
     [0] needs-to-visit(aircraft, city) <->
         (goal(at(aircraft, city)) |
          exists person [
                 [0] in(person, aircraft) &
                 goal(at(person, city)) ]) ]

// A plane is allowed to fly to a city if:
// 1: It's a goal, the plane is empty and no other persons need to be
//    transported.
// 2: The plane is carrying a passenger destined for the city.
// 3: A person in the city wants to leave, has not committed to leaving the city
//    already,
//    no other aircraft has committed to go to the city and either the plane
//    "needs-to-visit" the city or there doesn't exists any aircraft that
//    "needs-to-visit" the city and no aircraft that will need to drop someone
//    off in the city.
#control :name "planes-always-fly-to-goal"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] at(aircraft, city2)) &
                       ((goal(at(aircraft, city2)) &
                        !exists person [
                                [t] in(person, aircraft) ] &
                        [t] all-persons-at-their-destinations-or-in-planes) |
                        exists person [
                               [t] in(person, aircraft) &
                               goal(at(person, city2)) ] |
                        exists person [
                               ((([t] at(person, city2)) &
                                 !$committed(t+1, at(person, city2), false)) &
                                [t] in-wrong-city(person)) &
                               !exists aircraft2 [
                                      $committed(t+1,
                                                 at(aircraft2, city2),
                                                 true) ] &
                               (([t] needs-to-visit(aircraft, city2)) |
                                !exists aircraft2 [
                                        aircraft2 != aircraft &
                                        [t] needs-to-visit(aircraft2, city2) ] &
                                !exists aircraft2, person2 [
                                        aircraft2 != aircraft &
                                        $committed(t+1,
                                                   in(person2, aircraft2),
                                                   true) &
                                        goal(at(person2, city2)) ]) ] ]

// A plane is not allowed to fly to it's goal until all it's passengers that are
// destined for other cities have been delivered.
#control :name "planes-always-deliver-passengers-first"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] at(aircraft, city2)) &
                       (goal(at(aircraft, city2)) ->
                        forall person, city3 [
                               [t] in(person, aircraft) &
                               goal(at(person, city3)) -> city3 = city2 ]) ]]
```

```
// People only get on planes if they need to go somewhere.
// They only get on a plane if:
// 1: The plane already needs to visit the person's destination
// 2: There isn't any other plane that already needs to visit the person's
//    current location and goal.
#control :name "only-board-when-neccessary"
        forall t, person, aircraft [
                ([t] !in(person, aircraft) &
                 [t+1] in(person, aircraft)) ->
                exists city1, city2 [
                       [t] at(person, city1) &
                       goal(at(person, city2)) &
                       city1 != city2 &
                       ([t] needs-to-visit(aircraft, city2) |
                        !exists aircraft2 [
                                !at(aircraft, city2) &
                                needs-to-visit(aircraft2, city1) &
                                needs-to-visit(aircraft2, city2) ]) ] ]

// People only debark when they've arrived at their destination.
#control :name "only-debark-when-in-goal-city"
         forall t, person, aircraft [
                [t] in(person, aircraft) ->
                ([t+1] in(person, aircraft)) |
                exists city [
                       [t] at(aircraft, city) &
                       goal(at(person, city)) ] ]

#define [t] all-persons-at-their-destinations-or-in-planes:
        forall person, city [
               goal(at(person, city)) ->
               [t] at(person, city) |
               exists aircraft [ in(person, aircraft) ] ]

#define [t] in-wrong-city(thing):
        exists city1, city2 [
               [t] at(thing, city1) &
               goal(at(thing, city2)) &
               city1 != city2 ]
```

B.2 ZenoTravel SimpleTime

```
#domain integer :integer :lb 0 :ub 10000

#domain thing :elements {}
#domain aircraft :parent thing :elements {}
#domain person :parent thing :elements {}
#domain city :elements {}
#domain flevel :elements {}

#feature at(thing, city) :domain boolean :injective
#feature in(person, aircraft) :domain boolean :injective
#feature fuel-level(aircraft, flevel) :domain boolean :injective
#feature next(flevel, flevel) :domain boolean :double-injective

// Try to limit the number of refuels for each plane using a counter.
#feature number-of-refuels(aircraft) :domain integer :noinit
#feature needs-to-visit(aircraft, city) :domain boolean :noinit

#deffeature in-wrong-city(thing) :domain boolean
#deffeature all-persons-at-their-destinations :domain boolean
#deffeature reasonable-plane-location(aircraft, city) :domain boolean :uncached
```

```
// True when the aircraft has taken off but not yet reached it’s destination.
#feature flying-to(aircraft, city) :domain boolean :injective
// True while aircraft has started refueling but not finished.
#feature refueling(aircraft) :domain boolean
// True while person has started boarding but not finished.
#feature boarding(person, aircraft) :domain boolean

// Resource used as a semaphore to indicate that the aircraft is busy.
#resource sem_aircraft(aircraft) :domain integer :preference :none

#assert forall aircraft, person [ aircraft != person ]

// Initialize the semaphore.
#dom [0] forall aircraft [ $init(sem_aircraft(aircraft)) == 1 &
                           $minimum(sem_aircraft(aircraft)) == 0 &
                           $maximum(sem_aircraft(aircraft)) == 1 ]



#operator board(person, aircraft, city)
 :at t
 :precond       [t] at(person, city) &
                [t] at(aircraft, city)
 :duration      20
                // Borrow this aircrafts semaphore to prevent it from flying off
                // while the passenger boards. :borrow-nonex is non exclusive so
                // that only actions which needs the aircraft exlusively will be
                // forbidden, i.e. zoom.
 :resources     [+1,+20] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+1] at(person, city) := false,
                [+20] in(person, aircraft) := true,
                [+1,+19] boarding(person, aircraft) := true,
                [+20] boarding(person, aircraft) := false
 :context
   :forall      city2
   :precond     goal(at(person, city2))
   :effects     [+1] needs-to-visit(aircraft, city2) := true

#operator debark(person, aircraft, city)
 :at t
 :precond       [t] in(person, aircraft) &
                [t] at(aircraft, city)
 :duration      30
                // Don’t let the plane fly off.
 :resources     [+1,+30] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+1] in(person, aircraft) := false,
                [+30] at(person, city) := true
 :context
   :precond     !(goal(at(aircraft, city)) &
                  (exists person2 [
                          $committed(t+1, in(person2, aircraft), true) &
                          goal(!at(person2, city)) ] |
                   exists person2 [
                          [t] in(person2, aircraft) &
                          goal(!at(person2, city)) ]))
   :effects     [+1] needs-to-visit(aircraft, city) := false

//#operator fly(aircraft, city1, city2, flevel1, flevel2)
// :at t
// :precond     [t] at(aircraft, city1) &
//              [t] fuel-level(aircraft, flevel1) &
//              [t] next(flevel2, flevel1) &
```

```
//              city1 != city2 &
//              // Should be generated automatically:
//              !$committed(t+1, at(aircraft, city1), false)
// :context
//   :effects   [+1] at(aircraft, city1) := false,
//              [+1] flying-to(aircraft, city2) := true,
//              [+180] at(aircraft, city2) := true,
//              [+180] fuel-level(aircraft, flevel1) := false,
//              [+180] fuel-level(aircraft, flevel2) := true,
//              [+180] flying-to(aircraft, city2) := false

#operator zoom(aircraft, city1, city2, flevel1, flevel2, flevel3)
 :at t
 :precond       [t] at(aircraft, city1) &
                [t] fuel-level(aircraft, flevel1) &
                [t] next(flevel2, flevel1) &
                ([t] next(flevel3, flevel2)) &
                // Should be generated automatically:
                !$committed(t+1, at(aircraft, city1), false)
 :duration      100
                // Borrow this aircraft exclusively. None of the other operators
                // that need the resource can be used at the same time, i.e.
                // board and debark.
 :resources     [+1,+100] :borrow sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+1] at(aircraft, city1) := false,
                [+1] flying-to(aircraft, city2) := true,
                [+100] at(aircraft, city2) := true,
                [+100] fuel-level(aircraft, flevel1) := false,
                [+100] fuel-level(aircraft, flevel3) := true,
                [+100] flying-to(aircraft, city2) := false,
                // Reset the refuel counter so that the aircraft can refuel when
                // it has arrived.
                [+1] number-of-refuels(aircraft) := 0

#operator refuel(aircraft, city, flevel, flevel1)
 :at t
 :precond       [t] fuel-level(aircraft, flevel) &
                [t] next(flevel, flevel1) &
                [t] at(aircraft, city) &
                ([t] !refueling(aircraft)) &
                // An aircraft isn't allowed to refuel more than once unless
                // there exists a reasonable city for it to travel to.
                (([t] number-of-refuels(aircraft) < 1) |
                 exists city2 [
                        [t] city2 != city &
                        [t] reasonable-plane-location(aircraft, city2) ])
 :duration      73
                // Don't let the plane fly off.
 :resources     [+1,+73] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+73] fuel-level(aircraft, flevel) := false,
                [+73] fuel-level(aircraft, flevel1) := true,
                // Update the refuel counter.
                [+1] number-of-refuels(aircraft) :=
                     value(t, number-of-refuels(aircraft) + 1),
                [+1] refueling(aircraft) := true,
                [+73] refueling(aircraft) := false



// Initialize the refuel counter to zero for all planes.
#dom forall aircraft [ [0] number-of-refuels(aircraft) == 0 ]
```

```
#dom forall aircraft, city [
     [0] needs-to-visit(aircraft, city) <->
         (goal(at(aircraft, city)) |
          exists person [
                 [0] in(person, aircraft) &
                 goal(at(person, city)) ]) ]

// Only fly to cities which are reasonable-plane-location's for this aircraft.
#control :name "planes-always-fly-to-goal"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] flying-to(aircraft, city2)) &
                       [t] reasonable-plane-location(aircraft, city2) ]]

// A destination is reasonable for a plane if:
// 1: It's a goal and no other persons need to be transported.
// 2: The plane is carrying a passenger destined for the city.
// 3: A person in the city wants to leave, has not committed to leaving the city
//    already, no other aircraft has committed to go to the city and either the
//    plane "needs-to-visit" the city or there doesn't exists any aircraft that
//    "needs-to-visit" the city and no aircraft that will need to drop someone
//    off in the city.
#define [t] reasonable-plane-location(aircraft, city):
        ((goal(at(aircraft, city)) &
          [t] all-persons-at-their-destinations) |
         exists person [
                [t] in(person, aircraft) &
                goal(at(person, city)) ] |
         exists person [
                ((([t] at(person, city)) &
                  !$committed(t+1, at(person, city), false)) &
                 [t] in-wrong-city(person)) &
                !exists aircraft2 [
                        $committed(t+1, flying-to(aircraft2, city), true) ] &
                (([t] needs-to-visit(aircraft, city)) |
                 !exists aircraft2 [
                         aircraft2 != aircraft &
                         [t] needs-to-visit(aircraft2, city) ] &
                 (!exists aircraft2, person2 [
                          aircraft2 != aircraft &
                          $committed(t+1, boarding(person2, aircraft2), true) &
                          goal(at(person2, city)) ] |
                  !exists aircraft2, person2 [
                          aircraft2 != aircraft &
                          $committed(t+1, in(person2, aircraft2), true) &
                          goal(at(person2, city)) ])) ])

#control :name "planes-always-deliver-passengers-first"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] flying-to(aircraft, city2)) &
                       (goal(at(aircraft, city2)) ->
                        forall person, city3 [
                               [t] in(person, aircraft) &
                               goal(at(person, city3)) -> city3 = city2 ]) ]]

#control :name "only-board-when-neccessary"
        forall t, person, city1 [
```

```
               [t] at(person, city1) ->
               ([t+1] at(person, city1)) |
                exists aircraft, city2 [
                       [t] at(person, city1) &
                       goal(at(person, city2)) &
                       city1 != city2 &
                       ([t] needs-to-visit(aircraft, city2) |
                        !exists aircraft2 [
                                !at(aircraft, city2) &
                                needs-to-visit(aircraft2, city1) &
                                needs-to-visit(aircraft2, city2) ]) ] ]

#control :name "only-debark-when-in-goal-city"
         forall t, person, aircraft [
                [t] in(person, aircraft) ->
                ([t+1] in(person, aircraft)) |
                exists city [
                       [t] at(aircraft, city) &
                       goal(at(person, city)) ] ]

#define [t] all-persons-at-their-destinations:
        forall person, city [
               goal(at(person, city)) -> [t] at(person, city) ]

#define [t] in-wrong-city(thing):
        exists city1, city2 [
               [t] at(thing, city1) &
               goal(at(thing, city2)) &
               city1 != city2 ]
```

B.3 ZenoTravel Timed

```
// Tell the planner to divide all durations by one thousand before they are
// displayed.
#timescale 0.001

#domain integer :integer :lb 0 :ub 100000000

#domain thing :elements {}
#domain aircraft :parent thing :elements {}
#domain person :parent thing :elements {}
#domain city :elements {}

#feature at(thing, city) :domain boolean :injective
#feature in(person, aircraft) :domain boolean :injective
#feature fuel(aircraft) :domain integer
#feature distance(city, city) :domain integer :function
#feature slow-speed(aircraft) :domain integer :function
#feature fast-speed(aircraft) :domain integer :function
#feature slow-burn(aircraft) :domain integer :function
#feature fast-burn(aircraft) :domain integer :function
#feature capacity(aircraft) :domain integer :function
#feature refuel-rate(aircraft) :domain integer :function
// total-fuel-used was meant to be used in optimizing directives
// to the planners. Instead of completing the task as fast as
// possible, they could try to minimize fuel use. This feature
// was never used in the Time version, so it's commented out.
//#feature total-fuel-used :domain integer
#feature boarding-time :domain integer :function
#feature debarking-time :domain integer :function

#feature needs-to-visit(aircraft, city) :domain boolean :noinit
```

```
#deffeature in-wrong-city(thing) :domain boolean
#deffeature all-persons-at-their-destinations :domain boolean
#deffeature reasonable-plane-location(aircraft, city) :domain boolean :uncached
#deffeature fly-better-than-zoom(aircraft, city, city) :domain boolean

#feature flying-to(aircraft, city) :domain boolean :injective
#feature refueling(aircraft) :domain boolean
#feature boarding(person, aircraft) :domain boolean

#resource sem_aircraft(aircraft) :domain integer :preference :none

#assert forall aircraft, person [ aircraft != person ]

#dom [0] forall aircraft [ $init(sem_aircraft(aircraft)) == 1 &
                           $minimum(sem_aircraft(aircraft)) == 0 &
                           $maximum(sem_aircraft(aircraft)) == 1 ]



#operator board(person, aircraft, city)
 :at t
 :precond       [t] at(person, city) &
                [t] at(aircraft, city)
                // boarding-time specifies the time it takes to board a plane.
                // We multiply it by 1000 since we set the timescale to 0.001.
                // The resulting value is converted to the special time type
                // and the result of the somewhat cumbersome expression is
                // bound to the variable t2 for ease of reference.
 :duration      $maketime(value(t, 1000 * boarding-time)) :as t2
 :resources     [+1,+t2] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+1] at(person, city) := false,
                [+t2] in(person, aircraft) := true,
                [+1,+t2 - 1] boarding(person, aircraft) := true,
                [+t2] boarding(person, aircraft) := false
 :context
   :forall      city2
   :precond     goal(at(person, city2))
   :effects     [+1] needs-to-visit(aircraft, city2) := true

#operator debark(person, aircraft, city)
 :at t
 :precond       [t] in(person, aircraft) &
                [t] at(aircraft, city)
 :duration      $maketime(value(t, 1000 * debarking-time)) :as t2
 :resources     [+1,+t2] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects     [+1] in(person, aircraft) := false,
                [+t2] at(person, city) := true
 :context
   :precond     !(goal(at(aircraft, city)) &
                  (exists person2 [
                          $committed(t+1, in(person2, aircraft), true) &
                          goal(!at(person2, city)) ] |
                   exists person2 [
                          [t] in(person2, aircraft) &
                          goal(!at(person2, city)) ]))
   :effects     [+1] needs-to-visit(aircraft, city) := false

#operator fly(aircraft, city1, city2)
 :at t
 :precond       [t] at(aircraft, city1) &
                [t] fuel(aircraft) >= distance(city1, city2) *
                                      slow-burn(aircraft) &
```

```
               [t] fly-better-than-zoom(aircraft, city1, city2) &
               city1 != city2
:duration      $maketime(value(t, 1000 * distance(city1, city2) /
                                        slow-speed(aircraft))) :as t2
 :resources    [+1,+t2] :borrow sem_aircraft(aircraft) :amount 1
 :context
   :effects    [+1] at(aircraft, city1) := false,
               [+1] flying-to(aircraft, city2) := true,
               [+t2] at(aircraft, city2) := true,
               [+t2] flying-to(aircraft,  city2) := false,
//             [+t2] total-fuel-used := value(t, total-fuel-used +
//                                               distance(city1, city2) *
//                                               slow-burn(aircraft)),
               [+t2] fuel(aircraft) := value(t, fuel(aircraft) -
                                               distance(city1, city2) *
                                               slow-burn(aircraft))

#operator zoom(aircraft, city1, city2)
 :at t
 :precond      [t] at(aircraft, city1) &
               [t] fuel(aircraft) >= distance(city1, city2) *
                                     fast-burn(aircraft) &
               [t] !fly-better-than-zoom(aircraft, city1, city2) &
               city1 != city2
 :duration     $maketime(value(t, 1000 * distance(city1, city2) /
                                        fast-speed(aircraft))) :as t2
 :resources    [+1,+t2] :borrow sem_aircraft(aircraft) :amount 1
 :context
   :effects    [+1] at(aircraft, city1) := false,
               [+1] flying-to(aircraft, city2) := true,
               [+t2] at(aircraft, city2) := true,
               [+t2] flying-to(aircraft, city2) := false,
//             [+t2] total-fuel-used := value(t, total-fuel-used -
//                                               distance(city1, city2) *
//                                               fast-burn(aircraft)),
               [+t2] fuel(aircraft) := value(t, fuel(aircraft) -
                                               distance(city1, city2) *
                                               fast-burn(aircraft))

#operator refuel(aircraft, city)
 :at t
 :precond      [t] capacity(aircraft) > fuel(aircraft) &
               [t] at(aircraft, city) &
               ([t] !refueling(aircraft)) &
               exists city2 [
                      [t] reasonable-plane-location(aircraft, city2) &
                      city != city2 ]
 :duration     $maketime(value(t, 1000 *
                               (capacity(aircraft) - fuel(aircraft)) /
                               refuel-rate(aircraft))) :as t2
 :resources    [+1,+t2] :borrow-nonex sem_aircraft(aircraft) :amount 1
 :context
   :effects    [+1,+t2 - 1] refueling(aircraft) := true,
               [+t2] refueling(aircraft) := false,
               [+t2] fuel(aircraft) := value(t, capacity(aircraft))



#dom forall aircraft, city [
     [0] needs-to-visit(aircraft, city) <->
         (goal(at(aircraft, city)) |
          exists person [
                 [0] in(person, aircraft) &
                 goal(at(person, city)) ]) ]
```

```
#control :name "planes-always-fly-to-goal"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] flying-to(aircraft, city2)) &
                       [t] reasonable-plane-location(aircraft, city2) ]]

#define [t] reasonable-plane-location(aircraft, city):
        (goal(at(aircraft, city)) &
         [t] all-persons-at-their-destinations) |
        exists person [
               [t] in(person, aircraft) &
               goal(at(person, city)) ] |
        exists person [
               ((([t] at(person, city)) &
                 !$committed(t+1, at(person, city), false)) &
                [t] in-wrong-city(person)) &
               !exists aircraft2 [
                       $committed(t+1, flying-to(aircraft2, city), true) ] &
               (([t] needs-to-visit(aircraft, city)) |
                !exists aircraft2 [
                        aircraft2 != aircraft &
                        [t] needs-to-visit(aircraft2, city) ] &
                (!exists aircraft2, person2 [
                         aircraft2 != aircraft &
                         $committed(t+1, boarding(person2, aircraft2), true) &
                         goal(at(person2, city)) ] |
                 !exists aircraft2, person2 [
                         aircraft2 != aircraft &
                         $committed(t+1, in(person2, aircraft2), true) &
                         goal(at(person2, city)) ])) ]

#control :name "planes-always-deliver-passengers-first"
         forall t, aircraft, city [
                [t] at(aircraft, city) ->
                ([t+1] at(aircraft, city)) |
                exists city2 [
                       city2 != city &
                       ([t+1] flying-to(aircraft, city2)) &
                       (goal(at(aircraft, city2)) ->
                        forall person, city3 [
                               [t] in(person, aircraft) &
                               goal(at(person, city3)) -> city3 = city2 ]) ]]

#control :name "only-board-when-neccessary"
        forall t, person, city1 [
               [t] at(person, city1) ->
               ([t+1] at(person, city1)) |
                exists aircraft, city2 [
                       [t] at(person, city1) &
                       goal(at(person, city2)) &
                       city1 != city2 &
                       ([t] needs-to-visit(aircraft, city2) |
                        [t] !exists aircraft2 [
                                    !at(aircraft, city2) &
                                    needs-to-visit(aircraft2, city1) &
                                    needs-to-visit(aircraft2, city2) ]) ] ]

#control :name "only-debark-when-in-goal-city"
         forall t, person, aircraft [
                [t] in(person, aircraft) ->
```

```
               ([t+1] in(person, aircraft)) |
               exists city [
                      [t] at(aircraft, city) &
                      goal(at(person, city)) ] ]

#define [t] all-persons-at-their-destinations:
        forall person, city [
               goal(at(person, city)) -> [t] at(person, city) ]

#define [t] in-wrong-city(thing):
        exists city1, city2 [
               [t] at(thing, city1) &
               goal(at(thing, city2)) &
               city1 != city2 ]

// An aircraft needs to fly from city1 to city2.
// It should use fly instead of zoom if:
#define [t] fly-better-than-zoom(aircraft, city1, city2):
        // If it's faster wrt speed and refueling
        ([t] (10000 / slow-speed(aircraft) +
              10000 * slow-burn(aircraft) / refuel-rate(aircraft)) <
             (10000 / fast-speed(aircraft) +
              10000 * fast-burn(aircraft) / refuel-rate(aircraft))) |
        // If zoom is impossible across this distance
        ([t] distance(city1, city2) * fast-burn(aircraft) >
             capacity(aircraft)) |
        // If zoom has to refuel but fly doesn't
        ([t] fuel(aircraft) >= distance(city1, city2) * slow-burn(aircraft) &
             fuel(aircraft) < distance(city1, city2) * fast-burn(aircraft))
```

B.4 Depots STRIPS

```
#domain integer :integer :lb 0 :ub 1000

#domain object :elements {}
#domain place :parent object :elements {}
#domain locatable :parent object :elements {}
#domain depot :parent place :elements {}
#domain distributor :parent place :elements {}
#domain truck :parent locatable :elements {}
#domain hoist :parent locatable :elements {}
#domain surface :parent locatable :elements {}
#domain pallet :parent surface :elements {}
#domain crate :parent surface :elements {}

#feature at(locatable, place) :domain boolean :injective
// generalized-at is true for a crate and a place if the crate is at the place
// or being lifted by a hoist at the place.
#feature generalized-at(crate, place) :domain boolean :injective :noinit
                                      :secondary
#feature on(crate, surface) :domain boolean :double-injective
#feature in(crate, truck) :domain boolean :injective
#feature lifting(hoist, crate) :domain boolean :double-injective
#feature available(hoist) :domain boolean :secondary
#feature clear(surface) :domain boolean :secondary

#feature need-to-move(surface) :domain boolean :noinit :secondary
#deffeature need-to-move-init(surface) :domain boolean
#feature need-to-be-at(crate, place) :domain boolean :injective :noinit
                                    :secondary
#deffeature need-to-be-at-init(crate, place) :domain boolean
#feature goodtower(surface) :domain boolean :noinit
#deffeature goodtower-init(surface) :domain boolean
```

```
#resource sem_truck(truck) :domain integer :preference :none
#resource sem_crate(crate) :domain integer :preference :none

#assert forall t, crate, surface, hoist [
               [t] on(crate, surface) -> [t] !lifting(hoist, crate) ]
#assert forall t, crate, surface, hoist [
               [t] lifting(hoist, crate) -> [t] !on(crate, surface) ]
#assert forall t, crate, hoist, truck [
               [t] lifting(hoist, crate) -> [t] !in(crate, truck) ]

#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall crate [ $init(sem_crate(crate)) == 1 &
                        $minimum(sem_crate(crate)) == 0 &
                        $maximum(sem_crate(crate)) == 1 ]



#operator Lift(hoist, crate, surface, place)
 // :iterate changes the order in which TALplanner iterates over the argument
 // domains. Instead of trying all hoists on the first crate, the planner will
 // begin by trying all crates with the first hoist and so on.
 :iterate (crate, hoist, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] available(hoist) &
                [t] at(crate, place) &
                [t] on(crate, surface) &
                ([t] clear(crate))
 :resources     [+1] :borrow sem_crate(crate) :amount 1
 :context
   :effects     [+1] at(crate, place) := false,
                [+1] lifting(hoist, crate) := true,
                [+1] clear(crate) := false,
                [+1] available(hoist) := false,
                [+1] clear(surface) := true,
                [+1] on(crate, surface) := false,
                [+1] goodtower(crate) := false

#operator Drop(hoist, crate, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(surface, place) &
                [t] clear(surface) &
                ([t] lifting(hoist, crate)) &
                // Only create goodtowers.
                forall surface2 [
                       goal(on(crate, surface2)) -> surface2 = surface ] &
                !([t] need-to-move(surface)) &
                !exists crate2 [
                        crate2 != crate & goal(on(crate2, surface)) ] &
                [t] goodtower(surface)
 :resources     [+1] :borrow sem_crate(crate) :amount 1
 :context
   :effects     [+1] available(hoist) := true,
                [+1] lifting(hoist, crate) := false,
                [+1] at(crate, place) := true,
                [+1] clear(surface) := false,
                [+1] clear(crate) := true,
                [+1] on(crate, surface) := true,
                // The preconditions make sure that the tower
                // created is a goodtower.
```

```
                [+1] need-to-move(crate) := false,
                [+1] goodtower(crate) := true

#operator Load(hoist, crate, truck, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                ([t] lifting(hoist, crate))
 :resources     [+1] :borrow sem_truck(truck) :amount 1,
                [+1] :borrow sem_crate(crate) :amount 1
 :context
   :effects     [+1] lifting(hoist, crate) := false,
                [+1] in(crate, truck) := true,
                [+1] available(hoist) := true,
                [+1] generalized-at(crate, place) := false

#operator Unload(hoist, crate, truck, place)
 :iterate (hoist, truck, crate, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                [t] available(hoist) &
                [t] in(crate, truck)
 :resources     [+1] :borrow sem_truck(truck) :amount 1,
                [+1] :borrow sem_crate(crate) :amount 1
 :context
   :effects     [+1] in(crate, truck) := false,
                [+1] available(hoist) := false,
                [+1] lifting(hoist, crate) := true,
                [+1] generalized-at(crate, place) := true

#operator Drive(truck, place1, place2)
 :at t
 :precond       [t] at(truck, place1)
 :resources     [+1] :borrow sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(truck, place1) := false,
                [+1] at(truck, place2) := true



#dom [0] forall crate, place [ need-to-be-at(crate, place) <->
                               need-to-be-at-init(crate, place) ]
#dom [0] forall surface [ need-to-move(surface) <->
                          need-to-move-init(surface) ]
#dom [0] forall crate, place [ generalized-at(crate, place) <->
                               at(crate, place) ]
#dom [0] forall surface [ goodtower(surface) <->
                          goodtower-init(surface) ]

// A crate is a goodtower if the crate and the crates below it don't need to be
// moved to reach the goal.
#define [t] goodtower-init(surface1):
        ([t] !need-to-move(surface1)) &
        forall crate, surface2 [
               (surface1 = crate & [t] on(crate, surface2)) ->
               [t] goodtower-init(surface2) ]

// A crate will be moved if:
// 1. It's not on it's goal surface or
// 2. It's on top of another crate that needs to be moved or
// 3. It occupies a space needed by another crate.
#define [t] need-to-move-init(surface1):
        exists crate [
```

```
               crate = surface1 &
               (exists surface2 [
                       goal(on(crate, surface2)) &
                       [t] !on(crate, surface2) ] |
                exists crate2 [
                       ([t] on(crate, crate2) &
                           need-to-move-init(crate2)) ] |
                exists surface2 [
                       ([t] on(crate, surface2)) &
                       (exists crate3 [
                              goal(on(crate3, surface2)) &
                              crate3 != crate ]) ] )]

// Trucks stay at a location until there are no more crates there that can be
// loaded and moved.
#control :name "trucks-stay-until-everything-is-done"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                !exists crate [
                        [t] at(crate, place) &
                            clear(crate) &
                            need-to-move(crate) ] ]

// Trucks can only move to a location with a misplaced crate or to a location
// where a crate in the truck must be unloaded.
#control :name "trucks-always-move-to-goal"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                exists place2 [
                       place2 != place &
                       ([t+1] at(truck, place2)) &
                       // There is a crate at the destination that either should
                       // be at another location or should be stacked
                       // differently and there are no other trucks there to do
                       // the job.
                       (([t] exists crate, place3 [
                                   generalized-at(crate, place2) &
                                   ((need-to-be-at(crate, place3) &
                                     place2 != place3) |
                                    (need-to-be-at(crate, place2) &
                                     !goodtower(crate))) &
                                   !exists truck2 [
                                          at(truck2, place2) ] ]) |
                        // There is a crate in the truck that needs to be at the
                        // destination and the stack that it should be on is
                        // already finished and there are no other crates that
                        // should be at the destination that the truck could
                        // pick up first.
                        (exists crate [
                                ([t] in(crate, truck) &
                                     need-to-be-at(crate, place2)) &
                                forall crate2 [
                                       goal(on(crate, crate2)) ->
                                       [t] goodtower(crate2) ] ] &
                         !([t] exists crate, place3 [
                                     generalized-at(crate, place3) &
                                     need-to-be-at(crate, place2) &
                                     place3 != place2 ]))) ] ]

// A crate needs to be at a location if the goal puts the crate on a pallet
// there or on another crate which needs to be there.
#define [t] need-to-be-at-init(crate, place):
```

```
        exists pallet [
               goal(on(crate, pallet)) &
               [t] at(pallet, place) ] |
        exists crate2 [
               goal(on(crate, crate2)) &
               [t] need-to-be-at-init(crate2, place) ]

// Don't lift crates that are part of goodtowers.
#control :name "only-move-crates-when-necessary"
         forall t, crate, place1 [
                [t] at(crate, place1) ->
                ([t+1] at(crate, place1)) |
                [t] need-to-move(crate) ]

// Only unload a crate if it can be placed in it's goal position.
#control :name "only-unload-crates-when-necessary"
         forall t, crate, truck [
                [t] in(crate, truck) ->
                ([t+1] in(crate, truck)) |
                exists surface, place [
                       goal(on(crate, surface)) &
                       [t] at(surface, place) &
                       [t] at(truck, place) ] ]
```

B.5 Depots SimpleTime

```
#domain integer :integer :lb 0 :ub 1000

#domain object :elements {}
#domain place :parent object :elements {}
#domain locatable :parent object :elements {}
#domain depot :parent place :elements {}
#domain distributor :parent place :elements {}
#domain truck :parent locatable :elements {}
#domain hoist :parent locatable :elements {}
#domain surface :parent locatable :elements {}
#domain pallet :parent surface :elements {}
#domain crate :parent surface :elements {}

#feature at(locatable, place) :domain boolean :injective
#feature generalized-at(crate, place) :domain boolean :injective :noinit
:secondary
#feature on(crate, surface) :domain boolean :double-injective
#feature in(crate, truck) :domain boolean :injective
#feature lifting(hoist, crate) :domain boolean :injective
#feature available(hoist) :domain boolean :secondary
#feature clear(surface) :domain boolean :secondary
// The truck has started driving towards the place but not arrived yet.
#feature driving-to(truck, place) :domain boolean :injective

#feature need-to-move(surface) :domain boolean :noinit :secondary
#deffeature need-to-move-init(surface) :domain boolean
#feature need-to-be-at(crate, place) :domain boolean :injective :noinit
                                     :secondary
#deffeature need-to-be-at-init(crate, place) :domain boolean
#feature goodtower(surface) :domain boolean :noinit
#deffeature goodtower-init(surface) :domain boolean

#resource sem_truck(truck) :domain integer :preference :none
#resource sem_crate(crate) :domain integer :preference :none
#resource sem_hoist(hoist) :domain integer :preference :none

#assert forall t, crate, surface, hoist [
```

```
              [t] on(crate, surface) -> [t] !lifting(hoist, crate) ]
#assert forall t, crate, surface, hoist [
              [t] lifting(hoist, crate) -> [t] !on(crate, surface) ]
#assert forall t, crate, hoist, truck [
              [t] lifting(hoist, crate) -> [t] !in(crate, truck) ]

#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall crate [ $init(sem_crate(crate)) == 1 &
                        $minimum(sem_crate(crate)) == 0 &
                        $maximum(sem_crate(crate)) == 1 ]
#dom [0] forall hoist [ $init(sem_hoist(hoist)) == 1 &
                        $minimum(sem_hoist(hoist)) == 0 &
                        $maximum(sem_hoist(hoist)) == 1 ]



#operator Lift(hoist, crate, surface, place)
 :iterate (crate, hoist, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] available(hoist) &
                [t] at(crate, place) &
                [t] on(crate, surface) &
                [t] clear(crate)
 :duration      1
 :resources     [+1] :borrow sem_crate(crate) :amount 1,
                [+1] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1] at(crate, place) := false,
                [+1] lifting(hoist, crate) := true,
                [+1] clear(crate) := false,
                [+1] available(hoist) := false,
                [+1] clear(surface) := true,
                [+1] on(crate, surface) := false,
                [+1] goodtower(crate) := false

#operator Drop(hoist, crate, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(surface, place) &
                [t] clear(surface) &
                ([t] lifting(hoist, crate)) &
                // Only create goodtowers.
                forall surface2 [
                       goal(on(crate, surface2)) -> surface2 = surface ] &
                !([t] need-to-move(surface)) &
                !exists crate2 [
                        crate2 != crate &
                        goal(on(crate2, surface)) ] &
                [t] goodtower(surface)
 :duration      1
 :resources     [+1] :borrow sem_crate(crate) :amount 1,
                [+1] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1] available(hoist) := true,
                [+1] lifting(hoist, crate) := false,
                [+1] at(crate, place) := true,
                [+1] clear(surface) := false,
                [+1] clear(crate) := true,
                [+1] on(crate, surface) := true,
                // The preconditions make sure that the tower
                // created is a goodtower.
```

```
               [+1] need-to-move(crate) := false,
               [+1] goodtower(crate) := true

#operator Load(hoist, crate, truck, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                [t] lifting(hoist, crate)
 :duration      3
 :resources     [+1,+3] :borrow-nonex sem_truck(truck) :amount 1,
                [+1,+3] :borrow sem_crate(crate) :amount 1,
                [+1,+3] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+3] lifting(hoist, crate) := false,
                [+3] in(crate, truck) := true,
                [+3] available(hoist) := true,
                [+3] generalized-at(crate, place) := false

#operator Unload(hoist, crate, truck, place)
 :iterate (hoist, truck, crate, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                [t] available(hoist) &
                [t] in(crate, truck)
 :duration      4
 :resources     [+1,+4] :borrow-nonex sem_truck(truck) :amount 1,
                [+1,+4] :borrow sem_crate(crate) :amount 1,
                [+1,+4] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1] in(crate, truck) := false,
                [+1] available(hoist) := false,
                [+4] lifting(hoist, crate) := true,
                [+1] generalized-at(crate, place) := true

#operator Drive(truck, place1, place2)
 :at t
 :precond       [t] at(truck, place1)
 :duration      10
 :resources     [+1,+10] :borrow sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(truck, place1) := false,
                [+10] at(truck, place2) := true,
                [+1] driving-to(truck, place2) := true,
                [+10] driving-to(truck, place2) := false



#dom [0] forall crate, place [ need-to-be-at(crate, place) <->
                               need-to-be-at-init(crate, place) ]
#dom [0] forall surface [ need-to-move(surface) <->
                          need-to-move-init(surface) ]
#dom [0] forall crate, place [ generalized-at(crate, place) <->
                               at(crate, place) ]
#dom [0] forall surface [ goodtower(surface) <->
                          goodtower-init(surface) ]

#define [t] goodtower-init(surface1):
        ([t] !need-to-move(surface1)) &
        forall crate, surface2 [
               (surface1 = crate & [t] on(crate, surface2)) ->
               [t] goodtower-init(surface2) ]

#define [t] need-to-move-init(surface1):
```

```
        exists crate [
               crate = surface1 &
               (exists surface2 [
                      goal(on(crate, surface2)) &
                      [t] !on(crate, surface2) ] |
                exists crate2 [
                      ([t] on(crate, crate2) &
                           need-to-move-init(crate2)) ] |
                exists surface2 [
                      ([t] on(crate, surface2)) &
                      (exists crate3 [
                             goal(on(crate3, surface2)) &
                             crate3 != crate ]) ] )]

#control :name "trucks-stay-until-everything-is-done"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                !exists crate [
                        [t] at(crate, place) &
                            clear(crate) &
                            need-to-move(crate) ] ]

#control :name "trucks-always-move-to-goal"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                exists place2 [
                      place2 != place &
                      // The truck is on it's way to place2.
                      ([t+1] driving-to(truck, place2)) &
                      (([t] exists crate, place3 [
                                  generalized-at(crate, place2) &
                                  ((need-to-be-at(crate, place3) &
                                    place2 != place3) |
                                   (need-to-be-at(crate, place2) &
                                    !goodtower(crate))) &
                                  !exists truck2 [
                                         at(truck2, place2) ] ]) |
                       (exists crate [
                              ([t] in(crate, truck) &
                                   need-to-be-at(crate, place2)) &
                              forall crate2 [
                                     goal(on(crate, crate2)) ->
                                     [t] goodtower(crate2) ] ] &
                        !([t] exists crate, place3 [
                                    generalized-at(crate, place3) &
                                    need-to-be-at(crate, place2) &
                                    place3 != place2 ])) ] ]

#define [t] need-to-be-at-init(crate, place):
        exists pallet [
               goal(on(crate, pallet)) &
               [t] at(pallet, place) ] |
        exists crate2 [
               goal(on(crate, crate2)) &
               [t] need-to-be-at-init(crate2, place) ]

#control :name "only-move-crates-when-necessary"
         forall t, crate, place1 [
                [t] at(crate, place1) ->
                ([t+1] at(crate, place1)) |
                [t] need-to-move(crate) ]
```

```
#control :name "only-unload-crates-when-necessary"
         forall t, crate, truck [
                [t] in(crate, truck) ->
                ([t+1] in(crate, truck)) |
                exists surface, place [
                       goal(on(crate, surface)) &
                       [t] at(surface, place) &
                       [t] at(truck, place) ] ]
```

B.6 Depots Timed

```
#timescale 0.001

#domain integer :integer :lb 0 :ub 1000000
#domain object :elements {}
#domain place :parent object :elements {}
#domain locatable :parent object :elements {}
#domain depot :parent place :elements {}
#domain distributor :parent place :elements {}
#domain truck :parent locatable :elements {}
#domain hoist :parent locatable :elements {}
#domain surface :parent locatable :elements {}
#domain pallet :parent surface :elements {}
#domain crate :parent surface :elements {}

#feature at(locatable, place) :domain boolean :injective
#feature generalized-at(crate, place) :domain boolean :injective :noinit
#feature on(crate, surface) :domain boolean :double-injective
#feature in(crate, truck) :domain boolean :injective
#feature lifting(hoist, crate) :domain boolean :injective
#feature available(hoist) :domain boolean
#feature clear(surface) :domain boolean

#feature distance(place, place) :domain integer
#feature speed(truck) :domain integer
#feature weight(crate) :domain integer
#feature power(hoist) :domain integer

#feature driving-to(truck, place) :domain boolean :injective
#feature need-to-move(surface) :domain boolean :noinit :secondary
#deffeature need-to-move-init(surface) :domain boolean
#feature need-to-be-at(crate, place) :domain boolean :injective :noinit
                                     :secondary
#deffeature need-to-be-at-init(crate, place) :domain boolean
#feature goodtower(surface) :domain boolean :noinit
#deffeature goodtower-init(surface) :domain boolean

#resource sem_truck(truck) :domain integer :preference :none
#resource sem_crate(crate) :domain integer :preference :none
#resource sem_hoist(hoist) :domain integer :preference :none

#assert forall t, crate, surface, hoist [
               [t] on(crate, surface) -> [t] !lifting(hoist, crate) ]
#assert forall t, crate, surface, hoist [
               [t] lifting(hoist, crate) -> [t] !on(crate, surface) ]
#assert forall t, crate, hoist, truck [
               [t] lifting(hoist, crate) -> [t] !in(crate, truck) ]

#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall crate [ $init(sem_crate(crate)) == 1 &
                        $minimum(sem_crate(crate)) == 0 &
```

```
                        $maximum(sem_crate(crate)) == 1 ]
#dom [0] forall hoist [ $init(sem_hoist(hoist)) == 1 &
                        $minimum(sem_hoist(hoist)) == 0 &
                        $maximum(sem_hoist(hoist)) == 1 ]



#operator Lift(hoist, crate, surface, place)
 :iterate (crate, hoist, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] available(hoist) &
                [t] at(crate, place) &
                [t] on(crate, surface) &
                [t] clear(crate)
 :duration      1000
 :resources     [+1,+1000] :borrow sem_crate(crate) :amount 1,
                [+1,+1000] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1] at(crate, place) := false,
                [+1000] lifting(hoist, crate) := true,
                [+1] clear(crate) := false,
                [+1] available(hoist) := false,
                [+1000] clear(surface) := true,
                [+1] on(crate, surface) := false,
                [+1000] goodtower(crate) := false

#operator Drop(hoist, crate, surface, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(surface, place) &
                [t] clear(surface) &
                ([t] lifting(hoist, crate)) &
                // Only create goodtowers.
                forall surface2 [
                       goal(on(crate, surface2)) -> surface2 = surface ] &
                !([t] need-to-move(surface)) &
                !exists crate2 [
                        crate2 != crate &
                        goal(on(crate2, surface)) ] &
                [t] goodtower(surface))
 :duration      1000
 :resources     [+1,+1000] :borrow sem_crate(crate) :amount 1,
                [+1,+1000] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1000] available(hoist) := true,
                [+1000] lifting(hoist, crate) := false,
                [+1000] at(crate, place) := true,
                [+1] clear(surface) := false,
                [+1000] clear(crate) := true,
                [+1000] on(crate, surface) := true,
                [+1000] at(surface, place) := true, // TODO: prevail
                [+1] need-to-move(crate) := false,
                [+1] goodtower(crate) := true

#operator Load(hoist, crate, truck, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                [t] lifting(hoist, crate)
                // $max ensures that the duration is not less than one.
 :duration      $maketime(value(t, $max(1, 1000 * weight(crate) /
                                        power(hoist)))) :as t2
 :resources     [+1,+t2] :borrow-nonex sem_truck(truck) :amount 1,
```

```
                [+1,+t2] :borrow sem_crate(crate) :amount 1,
                [+1,+t2] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+t2] lifting(hoist, crate) := false,
                [+t2] in(crate, truck) := true,
                [+t2] available(hoist) := true,
                [+t2] generalized-at(crate, place) := false

#operator Unload(hoist, crate, truck, place)
 :iterate (hoist, truck, crate, place)
 :at t
 :precond       [t] at(hoist, place) &
                [t] at(truck, place) &
                [t] available(hoist) &
                [t] in(crate, truck)
 :duration      $maketime(value(t, $max(1, 1000 * weight(crate) /
power(hoist)))) :as t2
 :resources     [+1,+t2] :borrow-nonex sem_truck(truck) :amount 1,
                [+1,+t2] :borrow sem_crate(crate) :amount 1,
                [+1,+t2] :borrow sem_hoist(hoist) :amount 1
 :context
   :effects     [+1] in(crate, truck) := false,
                [+1] available(hoist) := false,
                [+t2] lifting(hoist, crate) := true,
                [+1] generalized-at(crate, place) := true

#operator Drive(truck, place1, place2)
 :at t
 :precond       [t] at(truck, place1) &
                place1 != place2
 :duration      $maketime(value(t, 1000 * distance(place1, place2) /
                                   speed(truck))) :as t2
 :resources     [+1,+t2] :borrow sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(truck, place1) := false,
                [+1] driving-to(truck, place2) := true,
                [+t2] at(truck, place2) := true,
                [+t2] driving-to(truck, place2) := false



#dom [0] forall crate, place [ need-to-be-at(crate, place) <->
                                 need-to-be-at-init(crate, place) ]
#dom [0] forall surface [ need-to-move(surface) <->
                           need-to-move-init(surface) ]
#dom [0] forall crate, place [ generalized-at(crate, place) <->
                                 at(crate, place) ]
#dom [0] forall surface [ goodtower(surface) <->
                           goodtower-init(surface) ]

#define [t] goodtower-init(surface1):
        ([t] !need-to-move(surface1)) &
        forall crate, surface2 [
               (surface1 = crate & [t] on(crate, surface2)) ->
               [t] goodtower-init(surface2) ]

#define [t] need-to-move-init(surface1):
        exists crate [
               crate = surface1 &
               (exists surface2 [
                       goal(on(crate, surface2)) &
                       [t] !on(crate, surface2) ] |
                exists crate2 [
                       ([t] on(crate, crate2) &
```

```
                        need-to-move-init(crate2)) ] |
               exists surface2 [
                      ([t] on(crate, surface2)) &
                      (exists crate3 [
                              goal(on(crate3, surface2)) &
                              crate3 != crate ]) ] )]

#control :name "trucks-stay-until-everything-is-done"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                !exists crate [
                        [t] at(crate, place) &
                            clear(crate) &
                            need-to-move(crate) ] ]

#control :name "trucks-always-move-to-goal"
         forall t, truck, place [
                ([t] at(truck, place)) ->
                ([t+1] at(truck, place)) |
                exists place2 [
                       place2 != place &
                       ([t+1] driving-to(truck, place2)) &
                       (([t] exists crate, place3 [
                                   generalized-at(crate, place2) &
                                   ((need-to-be-at(crate, place3) &
                                     place2 != place3) |
                                    (need-to-be-at(crate, place2) &
                                     !goodtower(crate))) &
                                   !exists truck2 [
                                          at(truck2, place2) ] ]) |
                        (exists crate [
                                ([t] in(crate, truck) &
                                     need-to-be-at(crate, place2)) &
                                forall crate2 [
                                       goal(on(crate, crate2)) ->
                                       [t] goodtower(crate2) ] ] &
                         !([t] exists crate, place3 [
                                     generalized-at(crate, place3) &
                                     need-to-be-at(crate, place2) &
                                     place3 != place2 ]))) ] ]

#define [t] need-to-be-at-init(crate, place):
        exists pallet [
               goal(on(crate, pallet)) &
               [t] at(pallet, place) ] |
        exists crate2 [
               goal(on(crate, crate2)) &
               [t] need-to-be-at-init(crate2, place) ]

#control :name "only-move-crates-when-necessary"
         forall t, crate, place1 [
                [t] at(crate, place1) ->
                ([t+1] at(crate, place1)) |
                [t] need-to-move(crate) ]

#control :name "only-unload-crates-when-necessary"
         forall t, crate, truck [
                [t] in(crate, truck) ->
                ([t+1] in(crate, truck)) |
                exists surface, place [
                       goal(on(crate, surface)) &
                       [t] at(surface, place) &
                       [t] at(truck, place) ] ]
```

B.7 DriverLog Strips

```
#domain integer :integer :lb 0 :ub 10000

#domain locatable :elements {}
#domain obj :parent locatable :elements {}
#domain location :elements {}
#domain truck :parent locatable :elements {}
#domain driver :parent locatable :elements {}

#feature at(locatable, location) :domain boolean :injective
#feature in(obj, truck) :domain boolean :injective
#feature driving(driver, truck) :domain boolean :double-injective
#feature link(location, location) :domain boolean
#feature path(location, location) :domain boolean
#feature empty(truck) :domain boolean :secondary

// Holds if driver has decided to walk to location (maybe via some other places)
#feature destination(driver, location) :domain boolean :injective
#deffeature reasonable-driver-location(driver, location)
            :domain boolean :uncached
#deffeature driving-distance-to-reasonable-destination(truck, location)
            :domain integer :uncached
#deffeature reasonable-truck-location(truck, location)
            :domain boolean :uncached
#deffeature all-objects-at-their-destinations :domain boolean
#deffeature all-nondriven-trucks-at-their-destinations-or-have-committed-drivers
            :domain boolean

// Define some location variables with more intuitive names than location1,
// location2 and so on.
#valuevar from, to, intermediate, dest :domain location

#distfeature driving-distance-between(from, to) :domain integer :link link
#distfeature walking-distance-between(from, to) :domain integer :link path

#mindistfeature mindist-driving :feature driving-distance-between
                                :domain integer
#mindistfeature mindist-walking :feature walking-distance-between
                                :domain integer

#resource objects-to-move-at(location) :domain integer :preference :none
#resource sem_truck(truck) :domain integer :preference :none
#resource sem_driver(driver) :domain integer :preference :none

#assert forall t, driver, location, truck [
               [t] driving(driver, truck) -> [t] !at(driver, location) ]
#assert forall t, driver, location, truck [
               [t] at(driver, location) -> [t] !driving(driver, truck) ]

#dom [0] forall location [ $init(objects-to-move-at(location)) ==
                           $sum(<obj>, [0] at(obj, location) &
                                       goal(!at(obj, location)), 1) &
                           $minimum(objects-to-move-at(location)) == 0 &
                           $maximum(objects-to-move-at(location)) == 9999 ]
#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall driver [ $init(sem_driver(driver)) == 1 &
                         $minimum(sem_driver(driver)) == 0 &
                         $maximum(sem_driver(driver)) == 1 ]
```

```
#operator board-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] at(driver, location) &
                ([t] empty(truck)) &
                // Don't board the truck if the goal can be reached by staying
                // put.
                ([t] all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers ->
                     !goal(at(driver, location)))
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location) := false,
                [+1] driving(driver, truck) := true,
                [+1] empty(truck) := false

#operator load-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                ([t] at(obj, location)) &
                // Don't load packages into a truck until we are sure that it
                // will have a driver.
                (([t] !empty(truck)) |
                 $committed(t+1, empty(truck), false))
 :resources     // One less object to load at this location.
                [+1] :consume objects-to-move-at(location) :amount 1,
                [+1] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(obj, location) := false,
                [+1] in(obj, truck) := true

#operator unload-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] in(obj, truck)
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] in(obj, truck) := false,
                [+1] at(obj, location) := true

#operator drive-truck(truck, location1, location2, driver)
 :iterate (truck, driver, location1, location2)
 :at t
 :precond       [t] at(truck, location1) &
                [t] driving(driver, truck) &
                ([t] link(location1, location2)) &
                location1 != location2 &
                // Don't drive if we're already at a reasonable location (the
                // feature returns 0) or if there are no reasonable locations
                // to go to (the feature returns infinity).
                ([t] driving-distance-to-reasonable-destination(truck,
location1) !== {0, 9999}) &
                // Only drive if it gets us closer to a reasonable location.
                [t] driving-distance-to-reasonable-destination(truck, location1)
>
                    driving-distance-to-reasonable-destination(truck, location2)
 :resources     [+1] :borrow sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(truck, location1) := false,
                [+1] at(truck, location2) := true
```

```
#operator disembark-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] driving(driver, truck)
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] driving(driver, truck) := false,
                [+1] at(driver, location) := true,
                [+1] empty(truck) := true

#operator walk-choosing-destination(driver, location1, location2, dest)
 // We have to make sure that the generated plan conforms with the given domain
 // specification, which only contains one walk operator. The planner will use
 // both walk-choosing-destination and walk-towards-destination but only print
 // "walk" when the final plan is output.
 :print walk(driver, location1, location2)
 // Begin by choosing a driver and his current location, then choose a
 // destination before deciding on the next immidiate step.
 :iterate (driver, location1, dest, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                // We have not already chosen a destination:
                !exists location3 [
                        [t] destination(driver, location3) ] &
                // Only choose reasonable destinations:
                ([t] reasonable-driver-location(driver, dest)) &
                // It's the closest reasonable destination:
                ([t] walking-distance-between(location1, dest) ==
                     mindist-walking(
                        location1,
                        to,
                        [t] reasonable-driver-location(driver, to))) &
                // We're not at a reasonable destination right now:
                ([t] !reasonable-driver-location(driver, location1)) &
                // Either noone else has already picked the destination or we
                // are walking to the final goal position:
                (!exists driver2 [
                         ([t] destination(driver2, dest)) |
                          $committed(t+1, destination(driver2, dest), true) ] |
                 [t] all-objects-at-their-destinations &
                     all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers) &
                // Having chosen a destination, we now need to select an
                // intermediary location2 that is on the way to the
                // destination.
                location1 != location2 &
                ([t] walking-distance-between(location1, dest) >
                     walking-distance-between(location2, dest))
 :resources     [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location1) := false,
                [+1] at(driver, location2) := true,
                [+1] destination(driver, dest) := true

#operator walk-towards-destination(driver, location1, location2)
 :print walk(driver, location1, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                // The step brings us closer to the destination:
                exists dest [
                       [t] destination(driver, dest) &
```

```
                       [t] walking-distance-between(location1, dest) >
                           walking-distance-between(location2, dest) ] &
                 location1 != location2
  :resources     [+1] :borrow sem_driver(driver) :amount 1
  :context
    :effects     [+1] at(driver, location1) := false,
                 [+1] at(driver, location2) := true
  :context
    :precond     // If we have arrived or the destination is no longer
                 // reasonable, make the driver free to choose another
                 // destination.
                 [t] destination(driver, location2) |
                 exists dest [
                        destination(driver, dest) &
                        !reasonable-driver-location(driver, dest) ]
    :effects     [+1] destination(driver, location2) := false



#define [t] driving-distance-to-reasonable-destination(truck, location):
        value(t, mindist-driving(location,
                                 to,
                                 [t] reasonable-truck-location(truck, to)))

// A location is reasonable for a truck if:
#define [t] reasonable-truck-location(truck, location):
        // The truck has objects to deliver there.
        exists obj [
               [t] in(obj, truck) &
               goal(at(obj, location)) ] |
        (([t] all-objects-at-their-destinations) &
         // There's a goal that the truck should be there.
         (goal(at(truck, location)) |
         // There's a goal that the driver should be there and no goal
         // preventing him from using the truck to drive there.
         (!goal(!at(truck, location)) &
          exists driver [
                 [t] driving(driver, truck) &
                 goal(at(driver, location)) ]))) |
        // There are objects to pick up and either we are already there or
        // no other trucks are already there or on their way.
        (([t] $available(objects-to-move-at(location)) !== 0) &
         (([t] at(truck, location)) |
          !exists truck2 [
                  truck2 != truck &
                  [t] !empty(truck2) &
                  [t] at(truck2, location) ] &
          !exists truck2 [
                  truck2 != truck &
                  ([t] !empty(truck2)) &
                  $committed(t+1, at(truck2, location), true) ]))

// A location is reasonable for a driver if:
#define [t] reasonable-driver-location(driver, location):
        // There are packages left to deliver and there is a truck without a
        // driver at the location.
        ([t] !all-objects-at-their-destinations) &
        exists truck [
               [t] at(truck, location) &
               ([t] empty(truck)) &
               !$committed(t+1, empty(truck), false) ] |
        // All packages have been delivered and either all trucks are at their
        // goals and the driver is heading for it's goal location or some trucks
        // still need to be driven to their goals and the driver is heading to
```

```
        // one of them.
        [t] all-objects-at-their-destinations &
        ([t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers &
         goal(at(driver, location)) |
         ([t] !all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers) &
         exists truck [
                [t] at(truck, location) &
                goal(!at(truck, location)) &
                !exists driver2 [
                        driver2 != driver &
                        [t] at(driver2, location) |
                        driving(driver2, truck) ] ])

// Only load packages if they aren't at their goal.
#control :name "only-load-when-necessary"
         forall t, obj, location1 [
                ([t] at(obj, location1)) &
                ([t+1] !at(obj, location1)) ->
                goal(!at(obj, location1)) ]

// Only unload packages at their goal.
#control :name "only-unload-when-necessary"
         forall t, obj, truck [
                [t] in(obj, truck) &
                ([t+1] !in(obj, truck)) ->
                exists location [
                       [t] at(truck, location) &
                       goal(at(obj, location)) ] ]

// Load and unload all packages that need to be loaded and unloaded before
// driving to another location.
#control :name "trucks-stay-until-everything-loaded-and-unloaded"
         forall t, truck, location [
                [t] at(truck, location) &
                [t+1] !at(truck, location) ->
                ([t] $available(objects-to-move-at(location)) == 0) &
                !exists obj [
                        [t] in(obj, truck) &
                        goal(at(obj, location)) ] ]

// Only board a truck if a drive-truck or a load-package will be possible.
// Loading packages are only possible after the driver has boarded the
// truck.
#control :name "only-board-when-necessary"
         forall t, driver, truck, location [
                [t] !driving(driver, truck) &
                    at(truck, location) ->
                [t+1] !driving(driver, truck) |
                ([t] driving-distance-to-reasonable-destination(truck, location)
                     !== {0, 9999}) |
                exists obj [
                       [t] at(obj, location) &
                       goal(!at(obj, location)) ] ]

// Don't disembark if there are packages loaded or being loaded into the truck
// that must be driven somewhere or if the truck has a goal to be somewhere
// else.
#control :name "only-disembark-when-necessary"
         forall t, driver, truck [
                [t] driving(driver, truck) ->
                ([t+1] driving(driver, truck)) |
                exists location [
```

```
                        ([t] at(truck, location)) &
                        (!exists obj [
                                goal(!at(obj, location)) &
                                $committed(t+1, in(obj, truck), true) |
                                [t] in(obj, truck) ] &
                         !goal(!at(truck, location))) ] ]

// Only disembark if you have driven to your goal or if there is another truck
// at the location that needs to be driven to its goal.
#control :name "only-disembark-when-you-have-a-goal"
         forall t, driver, truck [
                [t] driving(driver, truck) ->
                ([t+1] driving(driver, truck)) |
                exists location [
                       goal(at(driver, location)) ] |
                exists location, truck2 [
                       [t] at(truck, location) &
                           at(truck2, location) &
                           empty(truck2) &
                           !reasonable-truck-location(truck2, location) ] ]

// True when all packages are at their goal locations.
#define [t] all-objects-at-their-destinations:
        forall obj, location [
               goal(at(obj, location)) -> [t] at(obj, location) ]

// True when all trucks without drivers are at their goal locations or have a
// drivers committed to driving them there.
#define [t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers:
        forall truck, location [
               ([t] empty(truck) &
                goal(at(truck, location))) ->
               (([t] at(truck, location)) |
                exists location2, driver [
                       [t] at(truck, location2) &
                       ([t] destination(driver, location2) |
                        [t] at(driver, location2) &
                        !goal(at(driver, location2))) ]) ]
```

B.8 DriverLog SimpleTime

```
#domain integer :integer :lb 0 :ub 10000

#domain locatable :elements {}
#domain obj :parent locatable :elements {}
#domain location :elements {}
#domain truck :parent locatable :elements {}
#domain driver :parent locatable :elements {}

#feature at(locatable, location) :domain boolean :injective
#feature in(obj, truck) :domain boolean :injective
#feature driving(driver, truck) :domain boolean :double-injective
#feature link(location, location) :domain boolean
#feature path(location, location) :domain boolean
#feature empty(truck) :domain boolean :secondary

// True while a driver or a truck is going to a location but hasn't arrived yet.
#feature going-to(locatable, location) :domain boolean :injective
#feature destination(driver, location) :domain boolean :injective
#deffeature reasonable-driver-location(driver, location)
            :domain boolean :uncached
#deffeature driving-distance-to-reasonable-destination(truck, location)
```

```
            :domain integer :uncached
#deffeature reasonable-truck-location(truck, location)
            :domain boolean :uncached
#deffeature reasonable-truck-location-cached(truck, location) :domain boolean
#deffeature all-objects-at-their-destinations :domain boolean
#deffeature all-nondriven-trucks-at-their-destinations-or-have-committed-drivers
            :domain boolean

#valuevar from, to, intermediate, dest :domain location

#distfeature driving-distance-between(from, to) :domain integer :link link
#distfeature walking-distance-between(from, to) :domain integer :link path

#mindistfeature mindist-driving :feature driving-distance-between
                                :domain integer
#mindistfeature mindist-walking :feature walking-distance-between
                                :domain integer

#resource objects-to-move-at(location) :domain integer :preference :none
#resource sem_truck(truck) :domain integer :preference :none
#resource sem_driver(driver) :domain integer :preference :none

#assert forall t, driver, location, truck [
               [t] driving(driver, truck) -> [t] !at(driver, location) ]
#assert forall t, driver, location, truck [
               [t] at(driver, location) -> [t] !driving(driver, truck) ]
#assert forall t, driver, location, truck [
               [t] going-to(driver, location) -> [t] !driving(driver, truck) ]

#dom [0] forall location [ $init(objects-to-move-at(location)) ==
                           $sum(<obj>, [0] at(obj, location) &
                                       goal(!at(obj, location)), 1) &
                           $minimum(objects-to-move-at(location)) == 0 &
                           $maximum(objects-to-move-at(location)) == 9999 ]
#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall driver [ $init(sem_driver(driver)) == 1 &
                         $minimum(sem_driver(driver)) == 0 &
                         $maximum(sem_driver(driver)) == 1 ]



#operator board-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] at(driver, location) &
                [t] empty(truck) &
                ([t] all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers ->
                    !goal(at(driver, location)))
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location) := false,
                [+1] driving(driver, truck) := true,
                [+1] empty(truck) := false

#operator load-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] at(obj, location) &
                (([t] !empty(truck)) |
                 $committed(t+1, empty(truck), false))
```

```
 :duration      2
 :resources     [+1] :consume objects-to-move-at(location) :amount 1,
                [+1,+2] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(obj, location) := false,
                [+2] in(obj, truck) := true

#operator unload-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] in(obj, truck)
 :duration      2
 :resources     [+1,+2] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] in(obj, truck) := false,
                [+2] at(obj, location) := true

#operator drive-truck(truck, location1, location2, driver)
 :iterate (truck, driver, location1, location2)
 :at t
 :precond       [t] at(truck, location1) &
                [t] driving(driver, truck) &
                [t] link(location1, location2) &
                location1 != location2 &
                [t] driving-distance-to-reasonable-destination(truck, location1)
!== {0, 9999} &
                [t] driving-distance-to-reasonable-destination(truck, location1)
>
                    driving-distance-to-reasonable-destination(truck, location2)
 :duration      10
 :resources     [+1,+10] :borrow sem_truck(truck) :amount 1,
                [+1,+10] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(truck, location1) := false,
                [+10] at(truck, location2) := true,
                [+1] going-to(truck, location2) := true,
                [+10] going-to(truck, location2) := false

#operator disembark-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] driving(driver, truck)
 :duration      1
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] driving(driver, truck) := false,
                [+1] at(driver, location) := true,
                [+1] empty(truck) := true

#operator walk-choosing-destination(driver, location1, location2, dest)
 :print walk(driver, location1, location2)
 :iterate (driver, location1, dest, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                !exists location3 [
                        [t] destination(driver, location3) ] &
                ([t] reasonable-driver-location(driver, dest)) &
                ([t] walking-distance-between(location1, dest) ==
                     mindist-walking(
                        location1,
                        to,
                        [t] reasonable-driver-location(driver, to))) &
```

```
                ([t] !reasonable-driver-location(driver, location1)) &
                (!exists driver2 [
                        ([t] destination(driver2, dest)) |
                         $committed(t+1, destination(driver2, dest), true) ] |
                 [t] all-objects-at-their-destinations &
                     all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers) &
                location1 != location2 &
                ([t] walking-distance-between(location1, dest) >
                     walking-distance-between(location2, dest))
 :duration      20
 :resources     [+1,+20] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location1) := false,
                [+20] at(driver, location2) := true,
                [+1] destination(driver, dest) := true,
                [+1,+19] going-to(driver, location2) := true,
                [+20] going-to(driver, location2) := false

#operator walk-towards-destination(driver, location1, location2)
 :print walk(driver, location1, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                exists dest [
                       [t] destination(driver, dest) &
                       [t] walking-distance-between(location1, dest) >
                           walking-distance-between(location2, dest) ] &
                location1 != location2
 :duration      20
 :resources     [+1,+20] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location1) := false,
                [+20] at(driver, location2) := true,
                [+1] going-to(driver, location2) := true,
                [+1,+19] going-to(driver, location2) := true,
                [+20] going-to(driver, location2) := false
 :context
   :precond     [t] destination(driver, location2) |
                exists dest [
                       destination(driver, dest) &
                       !reasonable-driver-location(driver, dest) ]
   :effects     [+20] destination(driver, location2) := false



#define [t] driving-distance-to-reasonable-destination(truck, location):
        value(t, mindist-driving(location,
                                 to,
                                 [t] reasonable-truck-location(truck, to)))

#define [t] reasonable-truck-location(truck, location):
        ([t] reasonable-truck-location-cached(truck, location)) |
        (([t] $available(objects-to-move-at(location)) !== 0) &
         (([t] at(truck, location)) |
          !exists truck2 [
                  truck2 != truck &
                  [t] !empty(truck2) &
                  [t] at(truck2, location) ] &
          // Extra case when the truck is going to location but hasn't arrived.
          !exists truck2 [
                  truck2 != truck &
                  [t] !empty(truck2) &
                  [t] going-to(truck2, location) ] &
```

```
          !exists truck2 [
                  truck2 != truck &
                  ([t] !empty(truck2)) &
                  $committed(t+1, going-to(truck2, location), true)]))

#define [t] reasonable-truck-location-cached(truck, location):
        exists obj [
               [t] in(obj, truck) &
               goal(at(obj, location)) ] |
        ([t] all-objects-at-their-destinations &
         (goal(at(truck, location)) |
         (!goal(!at(truck, location)) &
          exists driver [
                 [t] driving(driver, truck) &
                 goal(at(driver, location)) ])))

#define [t] reasonable-driver-location(driver, location):
        ([t] !all-objects-at-their-destinations) &
        exists truck [
               [t] at(truck, location) &
               ([t] empty(truck)) &
               !$committed(t+1, empty(truck), false) ] |
        [t] all-objects-at-their-destinations &
        ([t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers &
         goal(at(driver, location)) |
         ([t] !all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers) &
         exists truck [
                [t] at(truck, location) &
                goal(!at(truck, location)) &
                !exists driver2 [
                        driver2 != driver &
                        (([t] at(driver2, location)) |
                         // Extra case when driver2 is going to location but
                         // hasn't arrived.
                         [t] going-to(driver2, location) |
                         driving(driver2, truck)) ] ])

#control :name "only-load-when-necessary"
         forall t, obj, location1 [
                ([t] at(obj, location1)) &
                ([t+1] !at(obj, location1)) ->
                goal(!at(obj, location1)) ]

#control :name "only-unload-when-necessary"
         forall t, obj, truck [
                ([t] in(obj, truck)) &
                ([t+1] !in(obj, truck)) ->
                exists location [
                        [t] at(truck, location) &
                        goal(at(obj, location)) ] ]

#control :name "trucks-stay-until-everything-loaded-and-unloaded"
         forall t, truck, location [
                [t] at(truck, location) &
                [t+1] !at(truck, location) ->
                ([t] $available(objects-to-move-at(location)) == 0) &
                !exists obj [
                        [t] in(obj, truck) &
                        goal(at(obj, location)) ] ]

#control :name "only-board-when-necessary"
         forall t, driver, truck, location [
```

```
               [t] !driving(driver, truck) &
                   at(truck, location) ->
               ([t+1] !driving(driver, truck)) |
               ([t] driving-distance-to-reasonable-destination(truck, location)
                    !== {0, 9999}) |
               exists obj [
                      [t] at(obj, location) &
                      goal(!at(obj, location)) ] ]

#control :name "only-disembark-when-necessary"
         forall t, driver, truck [
                [t] driving(driver, truck) ->
                ([t+1] driving(driver, truck)) |
                exists location [
                       ([t] at(truck, location)) &
                       (!exists obj [
                                goal(!at(obj, location)) &
                                $committed(t+1, in(obj, truck), true) |
                                [t] in(obj, truck) ] &
                        !goal(!at(truck, location))) ] ]

#control :name "only-disembark-when-you-have-a-goal"
         forall t, driver, truck [
                ([t] driving(driver, truck)) ->
                ([t+1] driving(driver, truck)) |
                exists location [
                       goal(at(driver, location)) ] |
                exists location, truck2 [
                       [t] at(truck, location) &
                           at(truck2, location) &
                           empty(truck2) &
                           !reasonable-truck-location(truck2, location) ] ]

#define [t] all-objects-at-their-destinations:
        forall obj, location [
               goal(at(obj, location)) -> [t] at(obj, location) ]

#define [t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers:
        forall truck, location [
               ([t] empty(truck) &
                goal(at(truck, location))) ->
               (([t] at(truck, location)) |
                exists location2, driver [
                       [t] at(truck, location2) &
                       ([t] destination(driver, location2) |
                        [t] at(driver, location2) &
                        !goal(at(driver, location2))) ]) ]
```

B.9 DriverLog Timed

```
#domain integer :integer :lb 0 :ub 10000

#domain locatable :elements {}
#domain obj :parent locatable :elements {}
#domain location :elements {}
#domain truck :parent locatable :elements {}
#domain driver :parent locatable :elements {}

#feature at(locatable, location) :domain boolean :injective
#feature in(obj, truck) :domain boolean :injective
#feature driving(driver, truck) :domain boolean :double-injective
#feature link(location, location) :domain boolean
```

```
#feature path(location, location) :domain boolean
#feature empty(truck) :domain boolean :secondary

// The walking distance between two locations, specified in the problem files.
#feature time-to-walk(location, location) :domain integer :function
// The driving distance between two locations, specified in the problem files.
#feature time-to-drive(location, location) :domain integer :function

#feature going-to(locatable, location) :domain boolean :injective
#feature destination(driver, location) :domain boolean :injective
#deffeature reasonable-driver-location(driver, location)
            :domain boolean :uncached
#deffeature driving-distance-to-reasonable-destination(truck, location)
            :domain integer :uncached
#deffeature reasonable-truck-location(truck, location)
            :domain boolean :uncached
#deffeature reasonable-truck-location-cached(truck, location) :domain boolean
#deffeature all-objects-at-their-destinations :domain boolean
#deffeature all-nondriven-trucks-at-their-destinations-or-have-committed-drivers
            :domain boolean

#valuevar from, to, intermediate, dest :domain location

#distfeature driving-distance-between(from, to)
             :domain integer :link link :cost time-to-drive
#distfeature walking-distance-between(from, to)
             :domain integer :link path :cost time-to-walk

#mindistfeature mindist-driving :feature driving-distance-between
                                :domain integer
#mindistfeature mindist-walking :feature walking-distance-between
                                :domain integer

#resource objects-to-move-at(location) :domain integer :preference :none
#resource sem_truck(truck) :domain integer :preference :none
#resource sem_driver(driver) :domain integer :preference :none

#assert forall t, driver, location, truck [
               [t] driving(driver, truck) -> [t] !at(driver, location) ]
#assert forall t, driver, location, truck [
               [t] at(driver, location) -> [t] !driving(driver, truck) ]
#assert forall t, driver, location, truck [
               [t] going-to(driver, location) -> [t] !driving(driver, truck) ]

#dom [0] forall location [ $init(objects-to-move-at(location)) ==
                           $sum(<obj>, [0] at(obj, location) &
                                       goal(!at(obj, location)), 1) &
                           $minimum(objects-to-move-at(location)) == 0 &
                           $maximum(objects-to-move-at(location)) == 9999 ]
#dom [0] forall truck [ $init(sem_truck(truck)) == 1 &
                        $minimum(sem_truck(truck)) == 0 &
                        $maximum(sem_truck(truck)) == 1 ]
#dom [0] forall driver [ $init(sem_driver(driver)) == 1 &
                         $minimum(sem_driver(driver)) == 0 &
                         $maximum(sem_driver(driver)) == 1 ]



#operator board-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] at(driver, location) &
                [t] empty(truck) &
```

```
               ([t] all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers ->
                    !goal(at(driver, location)))
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location) := false,
                [+1] driving(driver, truck) := true,
                [+1] empty(truck) := false

#operator load-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] at(obj, location) &
                (([t] !empty(truck)) |
                 $committed(t+1, empty(truck), false))
 :duration      2
 :resources     [+1] :consume objects-to-move-at(location) :amount 1,
                [+1,+2] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] at(obj, location) := false,
                [+2] in(obj, truck) := true

#operator unload-truck(obj, truck, location)
 :at t
 :precond       [t] at(truck, location) &
                [t] in(obj, truck)
 :duration      2
 :resources     [+1,+2] :borrow-nonex sem_truck(truck) :amount 1
 :context
   :effects     [+1] in(obj, truck) := false,
                [+2] at(obj, location) := true

#operator drive-truck(truck, location1, location2, driver)
 :iterate (truck, driver, location1, location2)
 :at t
 :precond       [t] at(truck, location1) &
                [t] driving(driver, truck) &
                [t] link(location1, location2) &
                location1 != location2 &
                ([t] driving-distance-to-reasonable-destination(truck,
location1) !== {0, 9999}) &
                // There is no cheaper road link that also reduce the value of
                // driving-distance-to-reasonable-destination.
                !exists location3 [
                        [t] link(location1, location3) &
                        [t] driving-distance-to-reasonable-destination(truck,
location3) +
                            driving-distance-between(location1, location3) <
                            driving-distance-to-reasonable-destination(truck,
location2) +
                            driving-distance-between(location1, location2) ]
 :duration      $maketime(value(t, time-to-drive(location1, location2))) :as t2
 :resources     [+1,+t2] :borrow sem_truck(truck) :amount 1,
                [+1,+t2] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(truck, location1) := false,
                [+t2] at(truck, location2) := true,
                [+1,+t2 - 1] going-to(truck, location2) := true,
                [+t2] going-to(truck, location2) := false

#operator disembark-truck(driver, truck, location)
 :at t
 :precond       [t] at(truck, location) &
```

```
                [t] driving(driver, truck)
 :duration      1
 :resources     [+1] :borrow-nonex sem_truck(truck) :amount 1,
                [+1] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] driving(driver, truck) := false,
                [+1] at(driver, location) := true,
                [+1] empty(truck) := true

#operator walk-choosing-destination(driver, location1, location2, dest)
 :print walk(driver, location1, location2)
 :iterate (driver, location1, dest, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                !exists location3 [
                        [t] destination(driver, location3) ] &
                ([t] reasonable-driver-location(driver, dest)) &
                ([t] walking-distance-between(location1, dest) ==
                     mindist-walking(
                        location1,
                        to,
                        [t] reasonable-driver-location(driver, to))) &
                ([t] !reasonable-driver-location(driver, location1)) &
                (!exists driver2 [
                         ([t] destination(driver2, dest)) |
                          $committed(t+1, destination(driver2, dest), true) ] |
                 [t] all-objects-at-their-destinations &
                     all-nondriven-trucks-at-their-destinations-or-have-
committed-drivers) &
                location1 != location2 &
                ([t] walking-distance-between(location1, dest) >
                     walking-distance-between(location2, dest))
 :duration      $maketime(value(t, time-to-walk(location1, location2))) :as t2
 :resources     [+1,+t2] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location1) := false,
                [+t2] at(driver, location2) := true,
                [+1] destination(driver, dest) := true,
                [+1,+t2 - 1] going-to(driver, location2) := true,
                [+t2] going-to(driver, location2) := false

#operator walk-towards-destination(driver, location1, location2)
 :print walk(driver, location1, location2)
 :at t
 :precond       [t] at(driver, location1) &
                ([t] path(location1, location2)) &
                exists dest [
                       [t] destination(driver, dest) &
                       [t] walking-distance-between(location1, dest) >
                           walking-distance-between(location2, dest) ] &
                location1 != location2
 :duration      $maketime(value(t, time-to-walk(location1, location2))) :as t2
 :resources     [+1,+t2] :borrow sem_driver(driver) :amount 1
 :context
   :effects     [+1] at(driver, location1) := false,
                [+t2] at(driver, location2) := true,
                [+1,+t2 - 1] going-to(driver, location2) := true,
                [+t2] going-to(driver, location2) := false
 :context
   :precond     [t] destination(driver, location2) |
                exists dest [
                       destination(driver, dest) &
                       !reasonable-driver-location(driver, dest) ]
```

```
   :effects     [+t2] destination(driver, location2) := false


#define [t] driving-distance-to-reasonable-destination(truck, location):
        value(t, mindist-driving(location,
                                 to,
                                 [t] reasonable-truck-location(truck, to)))

#define [t] reasonable-truck-location(truck, location):
        ([t] reasonable-truck-location-cached(truck, location)) |
        (([t] $available(objects-to-move-at(location)) !== 0) &
         (([t] at(truck, location)) |
          !exists truck2 [
                  truck2 != truck &
                  [t] !empty(truck2) &
                  [t] at(truck2, location) ] &
          !exists truck2 [
                  truck2 != truck &
                  [t] !empty(truck2) &
                  [t] going-to(truck2, location) ] &
          !exists truck2 [
                  truck2 != truck &
                  ([t] !empty(truck2)) &
                  $committed(t+1, going-to(truck2, location), true)]))

#define [t] reasonable-truck-location-cached(truck, location):
        exists obj [
               [t] in(obj, truck) &
               goal(at(obj, location)) ] |
        ([t] all-objects-at-their-destinations &
         (goal(at(truck, location)) |
         (!goal(!at(truck, location)) &
          exists driver [
                 [t] driving(driver, truck) &
                 goal(at(driver, location)) ])))

#define [t] reasonable-driver-location(driver, location):
        ([t] !all-objects-at-their-destinations) &
        exists truck [
               [t] at(truck, location) &
               ([t] empty(truck)) &
               !$committed(t+1, empty(truck), false) ] |
        [t] all-objects-at-their-destinations &
        ([t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers &
         goal(at(driver, location)) |
         ([t] !all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers) &
         exists truck [
                [t] at(truck, location) &
                goal(!at(truck, location)) &
                !exists driver2 [
                        driver2 != driver &
                        ([t] at(driver2, location) |
                             going-to(driver2, location) |
                             driving(driver2, truck)) ] ])

#control :name "only-load-when-necessary"
         forall t, obj, location1 [
                ([t] at(obj, location1)) &
                ([t+1] !at(obj, location1)) ->
                goal(!at(obj, location1)) ]
```

```
#control :name "only-unload-when-necessary"
         forall t, obj, truck [
                ([t] in(obj, truck)) &
                ([t+1] !in(obj, truck)) ->
                exists location [
                        [t] at(truck, location) &
                        goal(at(obj, location)) ] ]

#control :name "trucks-stay-until-everything-loaded-and-unloaded"
         forall t, truck, location [
                [t] at(truck, location) &
                [t+1] !at(truck, location) ->
                ([t] $available(objects-to-move-at(location)) == 0) &
                !exists obj [
                        [t] in(obj, truck) &
                        goal(at(obj, location)) ] ]

#control :name "only-board-when-necessary"
         forall t, driver, truck, location [
                [t] !driving(driver, truck) &
                    at(truck, location) ->
                ([t+1] !driving(driver, truck)) |
                ([t] driving-distance-to-reasonable-destination(truck, location)
                     !== {0, 9999}) |
                exists obj [
                       [t] at(obj, location) &
                       goal(!at(obj, location)) ] ]

#control :name "only-disembark-when-necessary"
         forall t, driver, truck [
                [t] driving(driver, truck) ->
                ([t+1] driving(driver, truck)) |
                exists location [
                       ([t] at(truck, location)) &
                       (!exists obj [
                                goal(!at(obj, location)) &
                                $committed(t+1, in(obj, truck), true) |
                                [t] in(obj, truck) ] &
                        !goal(!at(truck, location))) ] ]

#control :name "only-disembark-when-you-have-a-goal"
         forall t, driver, truck [
                ([t] driving(driver, truck)) ->
                ([t+1] driving(driver, truck)) |
                exists location [
                       goal(at(driver, location)) ] |
                exists location, truck2 [
                       [t] at(truck, location) &
                           at(truck2, location) &
                           empty(truck2) &
                           !reasonable-truck-location(truck2, location) ] ]

#define [t] all-objects-at-their-destinations:
        forall obj, location [
               goal(at(obj, location)) -> [t] at(obj, location) ]

#define [t] all-nondriven-trucks-at-their-destinations-or-have-committed-
drivers:
        forall truck, location [
               ([t] empty(truck) &
                goal(at(truck, location))) ->
               (([t] at(truck, location)) |
                exists location2, driver [
                       [t] at(truck, location2) &
```

```
                    ([t] destination(driver, location2) |
                     [t] at(driver, location2) &
                     !goal(at(driver, location2))) ]) ]
```

B.10 Rovers STRIPS

```
#domain integer :integer :lb 0 :ub 1000

#domain rover :elements {}
#domain waypoint :elements {}
#domain store :elements {}
#domain camera :elements {}
#domain mode :elements {}
#domain lander :elements {}
#domain objective :elements {}

#feature at(rover, waypoint) :domain boolean :injective
#feature at_lander(lander, waypoint) :domain boolean :injective
#feature can_traverse(rover, waypoint, waypoint) :domain boolean :function
#feature equipped_for_soil_analysis(rover) :domain boolean :function
#feature equipped_for_rock_analysis(rover) :domain boolean :function
#feature equipped_for_imaging(rover) :domain boolean :function
#feature empty(store) :domain boolean
#feature have_rock_analysis(rover, waypoint) :domain boolean
#feature have_soil_analysis(rover, waypoint) :domain boolean
#feature full(store) :domain boolean
#feature calibrated(camera, rover) :domain boolean :injective
#feature supports(camera, mode) :domain boolean :function
#feature available(rover) :domain boolean
#feature visible(waypoint, waypoint) :domain boolean :function
#feature have_image(rover, objective, mode) :domain boolean
#feature communicated_soil_data(waypoint) :domain boolean
#feature communicated_rock_data(waypoint) :domain boolean
#feature communicated_image_data(objective, mode) :domain boolean
#feature at_soil_sample(waypoint) :domain boolean
#feature at_rock_sample(waypoint) :domain boolean
#feature visible_from(objective, waypoint) :domain boolean :function
#feature store_of(store, rover) :domain boolean :injective
#feature calibration_target(camera, objective) :domain boolean :function
#feature on_board(camera, rover) :domain boolean :injective :function
#feature channel_free(lander) :domain boolean

#feature someone_has_rock_analysis(waypoint) :domain boolean
#feature someone_has_soil_analysis(waypoint) :domain boolean
#feature someone_has_image(objective, mode) :domain boolean

#deffeature roving-distance-to-reasonable-location(rover, waypoint)
            :domain integer
#deffeature reasonable-rover-location(rover, waypoint) :domain boolean
#distfeature roving-distance-between(rover, waypoint1, waypoint2)
             :domain integer :link can_traverse
#mindistfeature mindist-roving :feature roving-distance-between :domain integer

// Taking an image destroys the cameras calibration so you can't take two images
// with one camera at the same time.
#resource sem_take_image(camera) :domain integer :preference :none
// Don't take two images of the same objective with the same mode at the same
// time becausse it will generate two have_image(rover, objective, mode).
#resource sem_have_image(objective, mode) :domain integer :preference :none
// The lander has only one communication channel.
#resource sem_communicate_data(lander) :domain integer :preference :none
// Don't do two actions that fills the store of a rover at the same time.
#resource sem_store(store) :domain integer :preference :none
```

```
// Don’t sample the same rock or soil twice.
#resource sem_rock_sample(waypoint) :domain integer :preference :none
#resource sem_soil_sample(waypoint) :domain integer :preference :none

#dom [0] forall camera [
                $init(sem_take_image(camera)) == 1 &
                $minimum(sem_take_image(camera)) == 0 &
                $maximum(sem_take_image(camera)) == 1 ]
#dom [0] forall objective, mode [
                $init(sem_have_image(objective, mode)) == 1 &
                $minimum(sem_have_image(objective, mode)) == 0 &
                $maximum(sem_have_image(objective, mode)) == 1 ]
#dom [0] forall lander [
                $init(sem_communicate_data(lander)) == 1 &
                $minimum(sem_communicate_data(lander)) == 0 &
                $maximum(sem_communicate_data(lander)) == 1 ]
#dom [0] forall store [
                $init(sem_store(store)) == 1 &
                $minimum(sem_store(store)) == 0 &
                $maximum(sem_store(store)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_rock_sample(waypoint)) == 1 &
                $minimum(sem_rock_sample(waypoint)) == 0 &
                $maximum(sem_rock_sample(waypoint)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_soil_sample(waypoint)) == 1 &
                $minimum(sem_soil_sample(waypoint)) == 0 &
                $maximum(sem_soil_sample(waypoint)) == 1 ]



#operator sample_soil(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_soil_sample(waypoint) &
                [t] equipped_for_soil_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store)
 :resources     [+1] :borrow sem_store(store) :amount 1,
                [+1] :borrow sem_soil_sample(waypoint) :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+1] full(store) := true,
                [+1] have_soil_analysis(rover, waypoint) := true,
                [+1] someone_has_soil_analysis(waypoint) := true,
                [+1] at_soil_sample(waypoint) := false

#operator sample_rock(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_rock_sample(waypoint) &
                [t] equipped_for_rock_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store)
 :resources     [+1] :borrow sem_store(store) :amount 1,
                [+1] :borrow sem_rock_sample(waypoint) :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+1] full(store) := true,
                [+1] have_rock_analysis(rover, waypoint) := true,
                [+1] someone_has_rock_analysis(waypoint) := true,
                [+1] at_rock_sample(waypoint) := false

#operator take_image(rover, waypoint, objective, camera, mode)
```

```
 :at t
 :precond       [t] calibrated(camera, rover) &
                [t] on_board(camera, rover) &
                [t] equipped_for_imaging(rover) &
                [t] supports(camera, mode) &
                [t] visible_from(objective, waypoint) &
                ([t] at(rover, waypoint)) &
                // Don't take images that you already have.
                [t] !have_image(rover, objective, mode)
 :resources     [+1] :borrow sem_take_image(camera) :amount 1,
                [+1] :borrow sem_have_image(objective, mode) :amount 1
 :context
   :effects     [+1] have_image(rover, objective, mode) := true,
                [+1] someone_has_image(objective, mode) := true,
                [+1] calibrated(camera, rover) := false,
                // The rover must not have changed location while taking the
                // picture.
                [+1] at(rover, waypoint) := true

#operator communicate_soil_data(rover, lander, waypoint1, waypoint2, waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_soil_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                ([t] channel_free(lander)) &
                // Don't send data that you've already sent.
                [t] !communicated_soil_data(waypoint1)
 :resources     [+1] :borrow sem_communicate_data(lander) :amount 1
 :context
   :effects     //[+1] channel_free(lander) := false,
                //[+1] channel_free(lander) := true,
                [+1] communicated_soil_data(waypoint1) := true

#operator communicate_rock_data(rover, lander, waypoint1, waypoint2, waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_rock_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                ([t] channel_free(lander)) &
                // Don't send data that you've already sent.
                [t] !communicated_rock_data(waypoint1)
 :resources     [+1] :borrow sem_communicate_data(lander) :amount 1
 :context
   :effects     //[+1] channel_free(lander) := false,
                //[+1] channel_free(lander) := true,
                [+1] communicated_rock_data(waypoint1) := true

#operator communicate_image_data(rover, lander, objective, mode, waypoint2,
waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_image(rover, objective, mode) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                ([t] channel_free(lander)) &
                // Don't send data that you've already sent.
                [t] !communicated_image_data(objective, mode)
 :resources     [+1] :borrow sem_communicate_data(lander) :amount 1
 :context
```

```
   :effects     //[+1] channel_free(lander) := false,
                //[+1] channel_free(lander) := true,
                [+1] communicated_image_data(objective, mode) := true

#operator calibrate(rover, camera, objective, waypoint)
 :at t
 :precond       [t] equipped_for_imaging(rover) &
                [t] calibration_target(camera, objective) &
                [t] at(rover, waypoint) &
                [t] visible_from(objective, waypoint) &
                [t] on_board(camera, rover)
 :context
   :effects     [+1] calibrated(camera, rover) := true,
                // The rover must not have changed location while calibrating
                // the camera.
                [+1] at(rover, waypoint) := true

#operator drop(rover, store)
 :at t
 :precond       [t] store_of(store, rover) &
                [t] full(store)
 :context
   :effects     [+1] full(store) := false,
                [+1] empty(store) := true

#operator navigate(rover, waypoint1, waypoint2)
 :at t
 :precond       [t] can_traverse(rover, waypoint1, waypoint2) &
                [t] available(rover) &
                [t] at(rover, waypoint1) &
                ([t] visible(waypoint1, waypoint2)) &
                waypoint1 != waypoint2 &
                // Only navigate if we are not at a reasonable location, and
                // there exists a reasonable destination and navigating brings
                // us closer to it.
                [t] roving-distance-to-reasonable-location(rover, waypoint1)
                    !== {0, 999} &
                [t] roving-distance-to-reasonable-location(rover, waypoint1) >
                    roving-distance-to-reasonable-location(rover, waypoint2)
 :context
   :effects     [+1] at(rover, waypoint1) := false,
                [+1] at(rover, waypoint2) := true



// A waypoint is reasonable for a rover if:
#define [t] reasonable-rover-location(rover, waypoint):
        // We need to go get a rock sample.
        (goal(communicated_rock_data(waypoint)) &
         [t] at_rock_sample(waypoint) &
         [t] !someone_has_rock_analysis(waypoint) &
         [t] equipped_for_rock_analysis(rover)) |
        // We need to go get a soil sample.
        (goal(communicated_soil_data(waypoint)) &
         [t] at_soil_sample(waypoint) &
         [t] !someone_has_soil_analysis(waypoint) &
         [t] equipped_for_soil_analysis(rover)) |
        // We need to go take an image of an objective visible from it.
        exists mode, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] visible_from(objective, waypoint) &
               [t] !someone_has_image(objective, mode) &
               ([t] equipped_for_imaging(rover)) &
               exists camera [
```

```
                    [t] on_board(camera, rover) &
                    [t] supports(camera, mode) &
                    [t] calibrated(camera, rover) ]] |
        // We need to go calibrate a camera to take an image.
        exists mode, camera, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] !someone_has_image(objective, mode) &
               [t] supports(camera, mode) &
               [t] on_board(camera, rover) &
               [t] !calibrated(camera, rover) &
               [t] calibration_target(camera, objective) &
               [t] visible_from(objective, waypoint) ] |
        // We need to go send rock data to lander.
        exists waypoint2, waypoint3, lander [
               [t] have_rock_analysis(rover, waypoint2) &
               [t] !communicated_rock_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        // We need to go send soil data to lander.
        exists waypoint2, waypoint3, lander [
               [t] have_soil_analysis(rover, waypoint2) &
               [t] !communicated_soil_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        // We need to go send image data to lander.
        exists mode, objective, waypoint2, lander [
               [t] have_image(rover, objective, mode) &
               [t] !communicated_image_data(objective, mode) &
               [t] at_lander(lander, waypoint2) &
               [t] visible(waypoint2, waypoint) ]

#define [t] roving-distance-to-reasonable-location(rover, waypoint1):
       value(t, mindist-roving(rover,
                               waypoint1,
                               waypoint2,
                               [t] reasonable-rover-location(rover, waypoint2)))

// Only sample soil if it's a goal.
#control :name "only-sample-goal-soil"
         forall t, waypoint [
                [t] !someone_has_soil_analysis(waypoint) ->
                ([t+1] !someone_has_soil_analysis(waypoint)) |
                goal(communicated_soil_data(waypoint)) ]

// Only sample rock if it's a goal.
#control :name "only-sample-goal-rock"
         forall t, waypoint [
                [t] !someone_has_rock_analysis(waypoint) ->
                ([t+1] !someone_has_rock_analysis(waypoint)) |
                goal(communicated_rock_data(waypoint)) ]

// Only take image if it's a goal.
#control :name "only-take-goal-images"
         forall t, objective, mode, rover [
                [t] !someone_has_image(objective, mode) ->
                ([t+1] !someone_has_image(objective, mode)) |
                goal(communicated_image_data(objective, mode)) ]

// Only calibrate cameras that can be used to take images that are needed in
// the goal.
#control :name "only-calibrate-if-camera-needed"
         forall t, rover, camera [
                [t] !calibrated(camera, rover) ->
                ([t+1] !calibrated(camera, rover)) |
```

```
                exists objective, mode [
                       [t] supports(camera, mode) &
                       goal(communicated_image_data(objective, mode)) &
                       [t] !someone_has_image(objective, mode) ] ]

// Only empty a store if it's on a rover that needs it to sample soil or to
// sample rock.
#control :name "only-drop-if-neccessary"
         forall t, store [
                [t] full(store) ->
                ([t+1] full(store)) |
                exists rover [
                       ([t] store_of(store, rover)) &
                       exists waypoint [
                              goal(communicated_soil_data(waypoint)) &
                              [t] !someone_has_soil_analysis(waypoint) &
                              [t] at_soil_sample(waypoint) &
                              [t] at(rover, waypoint) &
                              [t] equipped_for_soil_analysis(rover) ] |
                       exists waypoint [
                              goal(communicated_rock_data(waypoint)) &
                              [t] !someone_has_rock_analysis(waypoint) &
                              [t] at_rock_sample(waypoint) &
                              [t] at(rover, waypoint) &
                              [t] equipped_for_rock_analysis(rover) ] ] ]
```

B.11 Rovers SimpleTime

```
#domain integer :integer :lb 0 :ub 1000

#domain rover :elements {}
#domain waypoint :elements {}
#domain store :elements {}
#domain camera :elements {}
#domain mode :elements {}
#domain lander :elements {}
#domain objective :elements {}

#feature at(rover, waypoint) :domain boolean :injective
#feature at_lander(lander, waypoint) :domain boolean :injective
#feature can_traverse(rover, waypoint, waypoint) :domain boolean :function
#feature equipped_for_soil_analysis(rover) :domain boolean :function
#feature equipped_for_rock_analysis(rover) :domain boolean :function
#feature equipped_for_imaging(rover) :domain boolean :function
#feature empty(store) :domain boolean
#feature have_rock_analysis(rover, waypoint) :domain boolean
#feature have_soil_analysis(rover, waypoint) :domain boolean
#feature full(store) :domain boolean
#feature calibrated(camera, rover) :domain boolean :injective
#feature supports(camera, mode) :domain boolean :function
#feature available(rover) :domain boolean
#feature visible(waypoint, waypoint) :domain boolean :function
#feature have_image(rover, objective, mode) :domain boolean
#feature communicated_soil_data(waypoint) :domain boolean
#feature communicated_rock_data(waypoint) :domain boolean
#feature communicated_image_data(objective, mode) :domain boolean
#feature at_soil_sample(waypoint) :domain boolean
#feature at_rock_sample(waypoint) :domain boolean
#feature visible_from(objective, waypoint) :domain boolean :function
#feature store_of(store, rover) :domain boolean :injective
#feature calibration_target(camera, objective) :domain boolean :function
#feature on_board(camera, rover) :domain boolean :function
#feature channel_free(lander) :domain boolean
```

```
// Someone has collected a rock or soil sample or an image or is in the
// process of doing so.
#feature someone_has_rock_analysis(waypoint) :domain boolean
#feature someone_has_soil_analysis(waypoint) :domain boolean
#feature someone_has_image(objective, mode) :domain boolean
// The camera has started calibrating but hasn't finished yet.
#feature calibrating(camera) :domain boolean

#deffeature roving-distance-to-reasonable-location(rover, waypoint)
            :domain integer
#deffeature reasonable-rover-location(rover, waypoint) :domain boolean
#distfeature roving-distance-between(rover, waypoint1, waypoint2)
             :domain integer :link can_traverse
#mindistfeature mindist-roving :feature roving-distance-between :domain integer

#resource sem_take_image(camera) :domain integer :preference :none
#resource sem_have_image(objective, mode) :domain integer :preference :none
#resource sem_communicate_data(lander) :domain integer :preference :none
#resource sem_store(store) :domain integer :preference :none
#resource sem_rock_sample(waypoint) :domain integer :preference :none
#resource sem_soil_sample(waypoint) :domain integer :preference :none
#resource sem_rover(rover) :domain integer :preference :none

#dom [0] forall camera [
                $init(sem_take_image(camera)) == 1 &
                $minimum(sem_take_image(camera)) == 0 &
                $maximum(sem_take_image(camera)) == 1 ]
#dom [0] forall objective, mode [
                $init(sem_have_image(objective, mode)) == 1 &
                $minimum(sem_have_image(objective, mode)) == 0 &
                $maximum(sem_have_image(objective, mode)) == 1 ]
#dom [0] forall lander [
                $init(sem_communicate_data(lander)) == 1 &
                $minimum(sem_communicate_data(lander)) == 0 &
                $maximum(sem_communicate_data(lander)) == 1 ]
#dom [0] forall store [
                $init(sem_store(store)) == 1 &
                $minimum(sem_store(store)) == 0 &
                $maximum(sem_store(store)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_rock_sample(waypoint)) == 1 &
                $minimum(sem_rock_sample(waypoint)) == 0 &
                $maximum(sem_rock_sample(waypoint)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_soil_sample(waypoint)) == 1 &
                $minimum(sem_soil_sample(waypoint)) == 0 &
                $maximum(sem_soil_sample(waypoint)) == 1 ]
#dom [0] forall rover [
                $init(sem_rover(rover)) == 1 &
                $minimum(sem_rover(rover)) == 0 &
                $maximum(sem_rover(rover)) == 1 ]



#operator sample_soil(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_soil_sample(waypoint) &
                [t] equipped_for_soil_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store)
 :duration      10
 :resources     [+1,+10] :borrow sem_store(store) :amount 1,
```

```
                [+1,+10] :borrow sem_soil_sample(waypoint) :amount 1,
                [+1,+10] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+10] full(store) := true,
                [+10] have_soil_analysis(rover, waypoint) := true,
                [+1] someone_has_soil_analysis(waypoint) := true,
                [+1] at_soil_sample(waypoint) := false

#operator sample_rock(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_rock_sample(waypoint) &
                [t] equipped_for_rock_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store)
 :duration      8
 :resources     [+1,+8] :borrow sem_store(store) :amount 1,
                [+1,+8] :borrow sem_rock_sample(waypoint) :amount 1,
                [+1,+8] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+8] full(store) := true,
                [+8] have_rock_analysis(rover, waypoint) := true,
                [+1] someone_has_rock_analysis(waypoint) := true,
                [+1] at_rock_sample(waypoint) := false

#operator take_image(rover, waypoint, objective, camera, mode)
 :at t
 :precond       [t] calibrated(camera, rover) &
                [t] on_board(camera, rover) &
                [t] equipped_for_imaging(rover) &
                [t] supports(camera, mode) &
                [t] visible_from(objective, waypoint) &
                [t] at(rover, waypoint) &
                [t] !someone_has_image(objective, mode)
 :duration      7
 :resources     [+1,+7] :borrow sem_take_image(camera) :amount 1,
                [+1,+7] :borrow sem_have_image(objective, mode) :amount 1,
                [+1,+7] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     [+7] have_image(rover, objective, mode) := true,
                [+1] someone_has_image(objective, mode) := true,
                [+7] calibrated(camera, rover) := false,
                [+7] at(rover, waypoint) := true

#operator communicate_soil_data(rover, lander, waypoint1, waypoint2, waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_soil_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                [t] channel_free(lander) &
                [t] !communicated_soil_data(waypoint1)
 :duration      10
 :resources     [+1,+10] :borrow sem_communicate_data(lander) :amount 1,
                [+1,+10] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     //[+1] channel_free(lander) := false,
                //[+10] channel_free(lander) := true,
                [+10] communicated_soil_data(waypoint1) := true

#operator communicate_rock_data(rover, lander, waypoint1, waypoint2, waypoint3)
```

```
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_rock_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                [t] channel_free(lander) &
                [t] !communicated_rock_data(waypoint1)
 :duration      10
 :resources     [+1,+10] :borrow sem_communicate_data(lander) :amount 1,
                [+1,+10] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     //[+1] channel_free(lander) := false,
                //[+10] channel_free(lander) := true,
                [+10] communicated_rock_data(waypoint1) := true

#operator communicate_image_data(rover, lander, objective, mode, waypoint2,
waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_image(rover, objective, mode) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                [t] channel_free(lander) &
                [t] !communicated_image_data(objective, mode)
 :duration      15
 :resources     [+1,+15] :borrow sem_communicate_data(lander) :amount 1,
                [+1,+10] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     //[+1] channel_free(lander) := false,
                //[+15] channel_free(lander) := true,
                [+15] communicated_image_data(objective, mode) := true

#operator calibrate(rover, camera, objective, waypoint)
 :at t
 :precond       [t] equipped_for_imaging(rover) &
                [t] calibration_target(camera, objective) &
                [t] at(rover, waypoint) &
                [t] visible_from(objective, waypoint) &
                [t] on_board(camera, rover) &
                [t] !calibrated(camera, rover) &
                [t] !calibrating(camera)
 :duration      5
 :resources     [+1,+5] :borrow-nonex sem_rover(rover) :amount 1
 :context
   :effects     [+5] calibrated(camera, rover) := true,
                [+5] at(rover, waypoint) := true,
                [+1] calibrating(camera) := true,
                [+5] calibrating(camera) := false

#operator drop(rover, store)
 :at t
 :precond       [t] store_of(store, rover) &
                [t] full(store)
 :context
   :effects     [+1] full(store) := false,
                [+1] empty(store) := true

#operator navigate(rover, waypoint1, waypoint2)
 :at t
 :precond       [t] can_traverse(rover, waypoint1, waypoint2) &
                [t] available(rover) &
                [t] at(rover, waypoint1) &
```

```
                [t] visible(waypoint1, waypoint2) &
                waypoint1 != waypoint2 &
                [t] roving-distance-to-reasonable-location(rover, waypoint1)
                    !== {0, 999} &
                [t] roving-distance-to-reasonable-location(rover, waypoint1) >
                    roving-distance-to-reasonable-location(rover, waypoint2)
 :duration      5
 :resources     [+1,+5] :borrow sem_rover(rover) :amount 1
 :context
   :effects     [+1] at(rover, waypoint1) := false,
                [+5] at(rover, waypoint2) := true



#define [t] reasonable-rover-location(rover, waypoint):
        (goal(communicated_rock_data(waypoint)) &
         [t] at_rock_sample(waypoint) &
         [t] !someone_has_rock_analysis(waypoint) &
         [t] equipped_for_rock_analysis(rover)) |
        (goal(communicated_soil_data(waypoint)) &
         [t] at_soil_sample(waypoint) &
         [t] !someone_has_soil_analysis(waypoint) &
         [t] equipped_for_soil_analysis(rover)) |
        exists mode, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] visible_from(objective, waypoint) &
               [t] !someone_has_image(objective, mode) &
               ([t] equipped_for_imaging(rover)) &
               exists camera [
                      [t] on_board(camera, rover) &
                      [t] supports(camera, mode) &
                      [t] calibrated(camera, rover) ]] |
        exists mode, camera, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] !someone_has_image(objective, mode) &
               [t] supports(camera, mode) &
               [t] on_board(camera, rover) &
               [t] !calibrated(camera, rover) &
               [t] calibration_target(camera, objective) &
               [t] visible_from(objective, waypoint) ] |
        exists waypoint2, waypoint3, lander [
               [t] have_rock_analysis(rover, waypoint2) &
               [t] !communicated_rock_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        exists waypoint2, waypoint3, lander [
               [t] have_soil_analysis(rover, waypoint2) &
               [t] !communicated_soil_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        exists mode, objective, waypoint2, lander [
               [t] have_image(rover, objective, mode) &
               [t] !communicated_image_data(objective, mode) &
               [t] at_lander(lander, waypoint2) &
               [t] visible(waypoint2, waypoint) ]

#define [t] roving-distance-to-reasonable-location(rover, waypoint1):
        value(t, mindist-roving(rover,
                                waypoint1,
                                waypoint2,
                                [t] reasonable-rover-location(rover,
waypoint2)))

#control :name "only-sample-goal-soil"
```

```
        forall t, waypoint [
               [t] !someone_has_soil_analysis(waypoint) ->
               ([t+1] !someone_has_soil_analysis(waypoint)) |
               goal(communicated_soil_data(waypoint)) ]

#control :name "only-sample-goal-rock"
        forall t, waypoint [
               [t] !someone_has_rock_analysis(waypoint) ->
               ([t+1] !someone_has_rock_analysis(waypoint)) |
               goal(communicated_rock_data(waypoint)) ]

#control :name "only-take-goal-images"
        forall t, objective, mode, rover [
               [t] !someone_has_image(objective, mode) ->
               ([t+1] !someone_has_image(objective, mode)) |
               goal(communicated_image_data(objective, mode)) ]

#control :name "only-calibrate-if-camera-needed"
        forall t, rover, camera [
               [t] !calibrating(camera) &
               ([t+1] calibrating(camera)) ->
               exists objective, mode [
                      [t] supports(camera, mode) &
                      goal(communicated_image_data(objective, mode)) &
                      [t] !someone_has_image(objective, mode) ]]

#control :name "only-drop-if-neccessary"
        forall t, store [
               [t] full(store) ->
               ([t+1] full(store)) |
               exists rover [
                      ([t] store_of(store, rover)) &
                      exists waypoint [
                             goal(communicated_soil_data(waypoint)) &
                             [t] !someone_has_soil_analysis(waypoint) &
                             [t] at_soil_sample(waypoint) &
                             [t] at(rover, waypoint) &
                             [t] equipped_for_soil_analysis(rover) ] |
                      exists waypoint [
                             goal(communicated_rock_data(waypoint)) &
                             [t] !someone_has_rock_analysis(waypoint) &
                             [t] at_rock_sample(waypoint) &
                             [t] at(rover, waypoint) &
                             [t] equipped_for_rock_analysis(rover) ] ]]
```

B.12 Rovers Timed

```
#timescale 0.001

#domain integer :integer :lb 0 :ub 1000000
#domain fixedpoint :fixedpoint :lb 0 :ub 200000 :decimals 4

#domain rover :elements {}
#domain waypoint :elements {}
#domain store :elements {}
#domain camera :elements {}
#domain mode :elements {}
#domain lander :elements {}
#domain objective :elements {}

#feature at(rover, waypoint) :domain boolean :injective
#feature at_lander(lander, waypoint) :domain boolean :injective
#feature can_traverse(rover, waypoint, waypoint) :domain boolean :function
```

```
#feature equipped_for_soil_analysis(rover) :domain boolean :function
#feature equipped_for_rock_analysis(rover) :domain boolean :function
#feature equipped_for_imaging(rover) :domain boolean :function
#feature empty(store) :domain boolean
#feature have_rock_analysis(rover, waypoint) :domain boolean
#feature have_soil_analysis(rover, waypoint) :domain boolean
#feature full(store) :domain boolean
#feature calibrated(camera, rover) :domain boolean :injective
#feature supports(camera, mode) :domain boolean :function
#feature available(rover) :domain boolean
#feature visible(waypoint, waypoint) :domain boolean :function
#feature have_image(rover, objective, mode) :domain boolean
#feature communicated_soil_data(waypoint) :domain boolean
#feature communicated_rock_data(waypoint) :domain boolean
#feature communicated_image_data(objective, mode) :domain boolean
#feature at_soil_sample(waypoint) :domain boolean
#feature at_rock_sample(waypoint) :domain boolean
#feature visible_from(objective, waypoint) :domain boolean :function
#feature store_of(store, rover) :domain boolean :injective
#feature calibration_target(camera, objective) :domain boolean :function
#feature on_board(camera, rover) :domain boolean :function
#feature channel_free(lander) :domain boolean

#feature in_sun(waypoint) :domain boolean :function
#feature energy(rover) :domain fixedpoint :function
#resource renergy(rover) :domain fixedpoint :preference :none
#feature recharge-rate(rover) :domain fixedpoint :function

#feature someone_has_rock_analysis(waypoint) :domain boolean
#feature someone_has_soil_analysis(waypoint) :domain boolean
#feature someone_has_image(objective, mode) :domain boolean
#feature calibrating(camera) :domain boolean
#feature recharging(rover) :domain boolean

#valuevar from, to :domain waypoint

#deffeature roving-distance-to-reasonable-location(rover, waypoint) :domain
integer
#deffeature roving-distance-to-recharge(rover, waypoint) :domain integer
#deffeature reasonable-rover-location(rover, waypoint) :domain boolean
#deffeature reasonable-rover-location-dont-care-energy(rover, waypoint) :domain
boolean
#distfeature roving-distance-between(rover, waypoint1, waypoint2) :domain
integer :link can_traverse
#mindistfeature mindist-roving :feature roving-distance-between :domain integer

#deffeature enough-energy-for-expedition(rover, from, to) :domain boolean
#deffeature have-enough-energy(rover, fixedpoint) :domain boolean :uncached

#resource sem_take_image(camera) :domain integer :preference :none
#resource sem_have_image(objective, mode) :domain integer :preference :none
#resource sem_communicate_data(lander) :domain integer :preference :none
#resource sem_store(store) :domain integer :preference :none
#resource sem_rock_sample(waypoint) :domain integer :preference :none
#resource sem_soil_sample(waypoint) :domain integer :preference :none
#resource sem_rover(rover) :domain integer :preference :none
#resource mutex_energy :domain integer :preference :none

#dom [0] forall camera [
                $init(sem_take_image(camera)) == 1 &
                $minimum(sem_take_image(camera)) == 0 &
                $maximum(sem_take_image(camera)) == 1 ]
#dom [0] forall objective, mode [
                $init(sem_have_image(objective, mode)) == 1 &
```

```
                $minimum(sem_have_image(objective, mode)) == 0 &
                $maximum(sem_have_image(objective, mode)) == 1 ]
#dom [0] forall lander [
                $init(sem_communicate_data(lander)) == 1 &
                $minimum(sem_communicate_data(lander)) == 0 &
                $maximum(sem_communicate_data(lander)) == 1 ]
#dom [0] forall store [
                $init(sem_store(store)) == 1 &
                $minimum(sem_store(store)) == 0 &
                $maximum(sem_store(store)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_rock_sample(waypoint)) == 1 &
                $minimum(sem_rock_sample(waypoint)) == 0 &
                $maximum(sem_rock_sample(waypoint)) == 1 ]
#dom [0] forall waypoint [
                $init(sem_soil_sample(waypoint)) == 1 &
                $minimum(sem_soil_sample(waypoint)) == 0 &
                $maximum(sem_soil_sample(waypoint)) == 1 ]
#dom [0] forall rover [
                $init(sem_rover(rover)) == 1 &
                $minimum(sem_rover(rover)) == 0 &
                $maximum(sem_rover(rover)) == 1 ]
#dom [0] forall rover [
                $init(renergy(rover)) == energy(rover) &
                $minimum(renergy(rover)) == 0.0 &
                $maximum(renergy(rover)) == 81.0 ]
#dom [0] $init(mutex_energy) == 1 &
         $minimum(mutex_energy) == 0 &
         $maximum(mutex_energy) == 1



#operator recharge(rover, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] in_sun(waypoint) &
                [t] $available(renergy(rover)) < 80.0 &
                [t] !recharging(rover)
 :duration      // Duration is: amount of energy charged / recharge rate of
                // rover. $max to make sure that the duration is not less than
                // one time step (which is not allowed).
                $maketime($cast(fixedpoint,
                                integer,
                                value(t, $max(1.0,
                                              1000.0 * (80.0 -
$available(renergy(rover))) / recharge-rate(rover))))) :as t2
 :resources     [+1,+t2] :borrow-nonex sem_rover(rover) :amount 1,
                // Recieved energy is: recharge duration * recharge rate of
                // rover.
                [+t2] :produce renergy(rover)
                      :amount $makevalue(fixedpoint, t2) *
                              recharge-rate(rover) / 1000.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1,+t2 - 1] recharging(rover) := true,
                [+t2] recharging(rover) := false

#operator sample_soil(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_soil_sample(waypoint) &
                [t] equipped_for_soil_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store) &
```

```
                [t] have-enough-energy(rover, 3.0)
 :duration      10000 :as t2
 :resources     [+1,+t2] :borrow sem_store(store) :amount 1,
                [+1,+t2] :borrow sem_soil_sample(waypoint) :amount 1,
                [+1,+t2] :borrow-nonex sem_rover(rover) :amount 1,
                [+1] :consume renergy(rover) :amount 3.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+t2] full(store) := true,
                [+t2] have_soil_analysis(rover, waypoint) := true,
                [+1] someone_has_soil_analysis(waypoint) := true,
                [+1] at_soil_sample(waypoint) := false

#operator sample_rock(rover, store, waypoint)
 :at t
 :precond       [t] at(rover, waypoint) &
                [t] at_rock_sample(waypoint) &
                [t] equipped_for_rock_analysis(rover) &
                [t] store_of(store, rover) &
                [t] empty(store) &
                [t] have-enough-energy(rover, 5.0)
 :duration      8000 :as t2
 :resources     [+1,+t2] :borrow sem_store(store) :amount 1,
                [+1,+t2] :borrow sem_rock_sample(waypoint) :amount 1,
                [+1,+t2] :borrow-nonex sem_rover(rover) :amount 1,
                [+1] :consume renergy(rover) :amount 5.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] empty(store) := false,
                [+t2] full(store) := true,
                [+t2] have_rock_analysis(rover, waypoint) := true,
                [+1] someone_has_rock_analysis(waypoint) := true,
                [+1] at_rock_sample(waypoint) := false

#operator take_image(rover, waypoint, objective, camera, mode)
 :at t
 :precond       [t] calibrated(camera, rover) &
                [t] on_board(camera, rover) &
                [t] equipped_for_imaging(rover) &
                [t] supports(camera, mode) &
                [t] visible_from(objective, waypoint) &
                [t] at(rover, waypoint) &
                ([t] !someone_has_image(objective, mode)) &
                [t] have-enough-energy(rover, 1.0)
 :duration      7000 :as t2
 :resources     [+1,+t2] :borrow sem_take_image(camera) :amount 1,
                [+1,+t2] :borrow sem_have_image(objective, mode) :amount 1,
                [+1,+t2] :borrow-nonex sem_rover(rover) :amount 1,
                [+1] :consume renergy(rover) :amount 1.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+t2] have_image(rover, objective, mode) := true,
                [+1] someone_has_image(objective, mode) := true,
                [+t2] calibrated(camera, rover) := false,
                [+t2] at(rover, waypoint) := true

#operator communicate_soil_data(rover, lander, waypoint1, waypoint2, waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_soil_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
```

```
                [t] channel_free(lander) &
                ([t] !communicated_soil_data(waypoint1)) &
                [t] have-enough-energy(rover, 4.0)
 :duration      10000 :as t2
 :resources     [+1,+t2] :borrow sem_communicate_data(lander) :amount 1,
                [+1] :consume renergy(rover) :amount 4.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] available(rover) := false,
                [+1] channel_free(lander) := false,
                [+t2] channel_free(lander) := true,
                [+t2] communicated_soil_data(waypoint1) := true,
                [+t2] available(rover) := true

#operator communicate_rock_data(rover, lander, waypoint1, waypoint2, waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_rock_analysis(rover, waypoint1) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                [t] channel_free(lander) &
                ([t] !communicated_rock_data(waypoint1)) &
                [t] have-enough-energy(rover, 4.0)
 :duration      10000 :as t2
 :resources     [+1,+t2] :borrow sem_communicate_data(lander) :amount 1,
                [+1] :consume renergy(rover) :amount 4.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] available(rover) := false,
                [+1] channel_free(lander) := false,
                [+t2] channel_free(lander) := true,
                [+t2] communicated_rock_data(waypoint1) := true,
                [+t2] available(rover) := true

#operator communicate_image_data(rover, lander, objective, mode, waypoint2,
waypoint3)
 :at t
 :precond       [t] at(rover, waypoint2) &
                [t] at_lander(lander, waypoint3) &
                [t] have_image(rover, objective, mode) &
                [t] visible(waypoint2, waypoint3) &
                [t] available(rover) &
                [t] channel_free(lander) &
                ([t] !communicated_image_data(objective, mode)) &
                [t] have-enough-energy(rover, 6.0)
 :duration      15000 :as t2
 :resources     [+1,+t2] :borrow sem_communicate_data(lander) :amount 1,
                [+1] :consume renergy(rover) :amount 6.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] available(rover) := false,
                [+1] channel_free(lander) := false,
                [+t2] channel_free(lander) := true,
                [+t2] communicated_image_data(objective, mode) := true,
                [+t2] available(rover) := true

#operator calibrate(rover, camera, objective, waypoint)
 :at t
 :precond       [t] equipped_for_imaging(rover) &
                [t] calibration_target(camera, objective) &
                [t] at(rover, waypoint) &
                [t] visible_from(objective, waypoint) &
                [t] on_board(camera, rover) &
```

```
                [t] !calibrated(camera, rover) &
                [t] !calibrating(camera) &
                [t] have-enough-energy(rover, 2.0)
 :duration      5000 :as t2
 :resources     [+1,+t2] :borrow-nonex sem_rover(rover) :amount 1,
                [+1] :consume renergy(rover) :amount 2.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+t2] calibrated(camera, rover) := true,
                [+t2] at(rover, waypoint) := true,
                [+1] calibrating(camera) := true,
                [+t2] calibrating(camera) := false

#operator drop(rover, store)
 :at t
 :precond       [t] store_of(store, rover) &
                [t] full(store)
 :duration      1000 :as t2
 :resources     [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] full(store) := false,
                [+t2] empty(store) := true

#operator navigate(rover, waypoint1, waypoint2)
 :at t
 :precond       [t] can_traverse(rover, waypoint1, waypoint2) &
                [t] available(rover) &
                [t] at(rover, waypoint1) &
                [t] visible(waypoint1, waypoint2) &
                waypoint1 != waypoint2 &
                [t] roving-distance-to-reasonable-location(rover, waypoint1) !==
{0, 999} &
                [t] roving-distance-to-reasonable-location(rover, waypoint1) >
                    roving-distance-to-reasonable-location(rover, waypoint2)
 :duration      5000 :as t2
 :resources     [+1,+t2] :borrow sem_rover(rover) :amount 1,
                [+1] :consume renergy(rover) :amount 8.0,
                [+1] :borrow mutex_energy :amount 1
 :context
   :effects     [+1] at(rover, waypoint1) := false,
                [+t2] at(rover, waypoint2) := true



// Taking energy into account, a location is reasonable for a rover if:
#define [t] reasonable-rover-location(rover, to):
        exists from [
               ([t] at(rover, from)) &
               // It's a place in the sun and either we don't have enough energy
               // to do one action and then go recharge or
               // there are no other reasonable locations that we can reach with
               // the available energy.
               (([t] in_sun(to) &
                 (([t] $available(renergy(rover)) <
                       $cast(integer,
                             fixedpoint,
                             value(t, roving-distance-to-recharge(rover, from)))
                       * 8.0 + 8.0) |
                  !exists waypoint3 [
                          waypoint3 != to &
                          [t] enough-energy-for-expedition(rover, from,
waypoint3) &
```

```
                        [t] reasonable-rover-location-dont-care-energy(rover,
waypoint3) ])) |
               // There is enough energy to go to place, do something and then
               // go recharge and that place is reasonable.
               ([t] enough-energy-for-expedition(rover, from, to) &
                [t] reasonable-rover-location-dont-care-energy(rover, to))) ]

// There is enough energy for the rover to go between from and to, perform
// at least one action and still have energy left to reach a recharge location.
#define [t] enough-energy-for-expedition(rover, from, to):
        [t] $cast(integer,
                  fixedpoint,
                  value(t, roving-distance-between(rover, from, to) +
                           roving-distance-to-recharge(rover, to)))
            * 8.0 + 8.0 <
            $available(renergy(rover))

// Without taking energy into account, a location is reasonable for a rover if
// it can perform some action there that helps achieve the goals.
#define [t] reasonable-rover-location-dont-care-energy(rover, waypoint):
        (goal(communicated_rock_data(waypoint)) &
         [t] at_rock_sample(waypoint) &
         [t] !someone_has_rock_analysis(waypoint) &
         [t] equipped_for_rock_analysis(rover)) |
        (goal(communicated_soil_data(waypoint)) &
         [t] at_soil_sample(waypoint) &
         [t] !someone_has_soil_analysis(waypoint) &
         [t] equipped_for_soil_analysis(rover)) |
        exists mode, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] visible_from(objective, waypoint) &
               [t] !someone_has_image(objective, mode) &
               ([t] equipped_for_imaging(rover)) &
               exists camera [
                      [t] on_board(camera, rover) &
                      [t] supports(camera, mode) &
                      [t] calibrated(camera, rover) ]] |
        exists mode, camera, objective [
               goal(communicated_image_data(objective, mode)) &
               [t] !someone_has_image(objective, mode) &
               [t] supports(camera, mode) &
               [t] on_board(camera, rover) &
               [t] !calibrated(camera, rover) &
               [t] calibration_target(camera, objective) &
               [t] visible_from(objective, waypoint) ] |
        exists waypoint2, waypoint3, lander [
               [t] have_rock_analysis(rover, waypoint2) &
               [t] !communicated_rock_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        exists waypoint2, waypoint3, lander [
               [t] have_soil_analysis(rover, waypoint2) &
               [t] !communicated_soil_data(waypoint2) &
               [t] at_lander(lander, waypoint3) &
               [t] visible(waypoint3, waypoint) ] |
        exists mode, objective, waypoint2, lander [
               [t] have_image(rover, objective, mode) &
               [t] !communicated_image_data(objective, mode) &
               [t] at_lander(lander, waypoint2) &
               [t] visible(waypoint2, waypoint) ]

#define [t] roving-distance-to-reasonable-location(rover, waypoint1):
       value(t, mindist-roving(rover,
                               waypoint1,
```

```
                              waypoint2,
                              [t] reasonable-rover-location(rover, waypoint2)))

// The distance to the closest waypoint that can be used to recharge the rover.
#define [t] roving-distance-to-recharge(rover, waypoint1):
        value(t, mindist-roving(rover,
                                waypoint1,
                                waypoint2,
                                [t] in_sun(waypoint2)))

// A rover has enough energy to do an action that consumes the amount of energy
// passed in the fixedpoint argument if it can do the action and still have
// enough energy to reach a recharge location.
#define [t] have-enough-energy(rover, fixedpoint):
        exists waypoint [
               [t] at(rover, waypoint) &
               [t] $cast(integer,
                         fixedpoint,
                         value(t, roving-distance-to-recharge(rover, waypoint)))
                         * 8.0 <
                         ($available(renergy(rover)) - fixedpoint) ]

#control :name "only-sample-goal-soil"
         forall t, waypoint [
                [t] !someone_has_soil_analysis(waypoint) ->
                ([t+1] !someone_has_soil_analysis(waypoint)) |
                goal(communicated_soil_data(waypoint)) ]

#control :name "only-sample-goal-rock"
         forall t, waypoint [
                [t] !someone_has_rock_analysis(waypoint) ->
                ([t+1] !someone_has_rock_analysis(waypoint)) |
                goal(communicated_rock_data(waypoint)) ]

#control :name "only-take-goal-images"
         forall t, objective, mode, rover [
                [t] !someone_has_image(objective, mode) ->
                ([t+1] !someone_has_image(objective, mode)) |
                goal(communicated_image_data(objective, mode)) ]

#control :name "only-calibrate-if-camera-needed"
         forall t, rover, camera [
                [t] !calibrating(camera) &
                ([t+1] calibrating(camera)) ->
                exists objective, mode [
                       [t] supports(camera, mode) &
                       goal(communicated_image_data(objective, mode)) &
                       [t] !someone_has_image(objective, mode) ] ]

#control :name "only-drop-if-neccessary"
         forall t, store [
                [t] full(store) ->
                ([t+1] full(store)) |
                exists rover [
                       ([t] store_of(store, rover)) &
                       exists waypoint [
                              goal(communicated_soil_data(waypoint)) &
                              [t] !someone_has_soil_analysis(waypoint) &
                              [t] at_soil_sample(waypoint) &
                              [t] at(rover, waypoint) &
                              [t] equipped_for_soil_analysis(rover) ] |
                       exists waypoint [
                              goal(communicated_rock_data(waypoint)) &
                              [t] !someone_has_rock_analysis(waypoint) &
```

```
                              [t] at_rock_sample(waypoint) &
                              [t] at(rover, waypoint) &
                              [t] equipped_for_rock_analysis(rover) ] ] ]
```

B.13 Satellite STRIPS

```
#domain integer :integer :lb 0 :ub 20

#domain satellite :elements {}
#domain direction :elements {}
#domain instrument :elements {}
#domain mode :elements {}

#valuevar old_direction, new_direction :domain direction

#feature on_board(instrument, satellite) :domain boolean :injective
#feature supports(instrument, mode) :domain boolean :function
#feature pointing(satellite, direction) :domain boolean :injective
#feature power_avail(satellite) :domain boolean
#feature power_on(instrument) :domain boolean
#feature calibrated(instrument) :domain boolean
#feature have_image(direction, mode) :domain boolean
#feature calibration_target(instrument, direction) :domain boolean :function

#deffeature goal_direction(satellite, direction) :domain boolean
#deffeature all_images_collected :domain boolean
#deffeature take_image_possible(satellite, direction) :domain boolean
#deffeature usefulness(instrument) :domain integer
#deffeature mode_needed_for_goal(mode) :domain boolean



#operator take_image(satellite, direction, instrument, mode)
 :at t
 :precond       [t] calibrated(instrument) &
                [t] on_board(instrument, satellite) &
                [t] supports(instrument, mode) &
                [t] power_on(instrument) &
                ([t] pointing(satellite, direction)) &
                // Don't take images that we already have.
                ([t] !have_image(direction, mode)) &
                !$committed(t+1, have_image(direction, mode), true)
 :context
   :effects     [+1] have_image(direction, mode) := true,
                // The satellite must not change direction while the picture is
                // being taken.
                [+1] pointing(satellite, direction) := true

#operator switch_on(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] power_avail(satellite)
 :context
   :effects     [+1] power_on(instrument) := true,
                [+1] calibrated(instrument) := false,
                [+1] power_avail(satellite) := false

#operator turn_to(satellite, new_direction, old_direction)
 :at t
 :precond       [t] pointing(satellite, old_direction) &
                [t] new_direction != old_direction
 :context
   :effects     [+1] pointing(satellite, new_direction) := true,
```

```
               [+1] pointing(satellite, old_direction) := false

#operator switch_off(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] power_on(instrument)
 :context
   :effects     [+1] power_on(instrument) := false,
                [+1] power_avail(satellite) := true

#operator calibrate(satellite, instrument, direction)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] calibration_target(instrument, direction) &
                [t] pointing(satellite, direction) &
                [t] power_on(instrument) &
                [t] !calibrated(instrument)
 :context
   :effects     [+1] calibrated(instrument) := true



#control :name "only-take-pictures-of-goals"
         forall t, direction, mode [
                [t] !have_image(direction, mode) &
                [t+1] have_image(direction, mode) ->
                goal(have_image(direction, mode)) ]

#control :name "only-point-in-goal-directions"
         forall t, satellite, direction [
                [t] pointing(satellite, direction) ->
                ([t+1] pointing(satellite, direction)) |
                exists new_direction [
                       [t+1] pointing(satellite, new_direction) &
                       ([t] goal_direction(satellite, new_direction)) ] ]

// It is useful for the satellite to point in the direction if:
#define [t] goal_direction(satellite, direction):
        // An image in the direction is possible and is a goal.
        ([t] take_image_possible(satellite, direction)) |
        // We need to calibrate an instrument.
        exists instrument [
                [t] calibration_target(instrument, direction) &
                [t] on_board(instrument, satellite) &
                [t] !calibrated(instrument) &
                [t] power_on(instrument) ] |
        // Pointing in the direction is a goal and all images have been
        // collected.
        (goal(pointing(satellite, direction)) &
         [t] all_images_collected)

// If the satellite points in the direction, the instrumentation is ready to
// take a picture of it, the picture has not been taken and taking the picture
// is a goal.
#define [t] take_image_possible(satellite, direction):
        exists mode [
               goal(have_image(direction, mode)) &
               !$committed(t+1, have_image(direction, mode), true) &
               ([t] !have_image(direction, mode)) &
               exists instrument [
                      [t] supports(instrument, mode) &
                      [t] on_board(instrument, satellite) &
                      [t] power_on(instrument) &
                      [t] calibrated(instrument) ]]
```

```
#define [t] all_images_collected:
        !exists direction, mode [
                goal(have_image(direction, mode)) &
                [t] !have_image(direction, mode) ]

// Don't turn towards a direction that another satellite has already decided
// to turn to.
#control :name "don't-all-point-in-same-direction"
         forall t, satellite, direction [
                [t] !pointing(satellite, direction) ->
                ([t+1] !pointing(satellite, direction)) |
                !exists satellite2 [
                      $committed(t+1, pointing(satellite2, direction), true) ] ]

// An instrument is more useful the more imaging modes it supports that are
// needed to fulfill the goals.
#define [t] usefulness(instrument):
        value(t, $sum(<mode>,
                      [t] supports(instrument, mode) &
                          mode_needed_for_goal(mode),
                      1))

// A mode is needed if at least one goal is to have an image using that mode
// and we have not yet taken that image.
#define [t] mode_needed_for_goal(mode):
        exists direction [
               goal(have_image(direction, mode)) &
               [t] !have_image(direction, mode) ]

// Only power on an instrument if there are no other instruments that are more
// useful or that are already switched on.
#control :name "use-the-most-useful-instrument"
         forall t, instrument [
                [t] !power_on(instrument) ->
                ([t+1] !power_on(instrument)) |
                ([t] usefulness(instrument) > 0) &
                !exists satellite, instrument2 [
                        [t] usefulness(instrument2) > usefulness(instrument) &
                        [t] on_board(instrument, satellite) &
                        [t] on_board(instrument2, satellite) ] ]

// Only power off an instrument if it is no longer of any use.
#control :name "don't-switch-instrument-off-if-you-don't-have-to"
         forall t, instrument [
                [t] power_on(instrument) ->
                ([t+1] power_on(instrument)) |
                !exists mode [
                        [t] supports(instrument, mode) &
                        [t] mode_needed_for_goal(mode) ] ]
```

B.14 Satellite SimpleTime

```
#domain integer :integer :lb 0 :ub 2000

#domain satellite :elements {}
#domain direction :elements {}
#domain instrument :elements {}
#domain mode :elements {}

#valuevar old_direction, new_direction :domain direction

#feature on_board(instrument, satellite) :domain boolean :function
```

```
#feature supports(instrument, mode) :domain boolean :function
#feature pointing(satellite, direction) :domain boolean :injective
#feature power_avail(satellite) :domain boolean
#feature power_on(instrument) :domain boolean
#feature calibrated(instrument) :domain boolean
#feature have_image(direction, mode) :domain boolean
#feature calibration_target(instrument, direction) :domain boolean :function

// Satellite has started turning but not finished.
#feature turning_towards(satellite, direction) :domain boolean :injective
// Someone has taken the image or is in the process of taking it.
#feature have_image_generalized(direction, mode) :domain boolean
// The instrument is powered on or powering on.
#feature power_on_generalized(instrument) :domain boolean
// The instrument has started calibrating but not finished.
#feature calibrating(instrument) :domain boolean

#deffeature goal_direction(satellite, direction) :domain boolean
#deffeature all_images_collected :domain boolean
#deffeature take_image_possible(satellite, direction) :domain boolean
#deffeature usefulness(instrument) :domain integer
#deffeature mode_needed_for_goal(mode) :domain boolean

// Don't power up two instruments on the same satellite at the same time.
#feature sem_power_on(satellite, instrument) :domain boolean :injective
// If an instrument is power_on it must also be power_on_generalized.
#assert forall t, instrument [
               [t] power_on(instrument) -> power_on_generalized(instrument) ]

#operator take_image(satellite, direction, instrument, mode)
 :at t
 :precond       [t] calibrated(instrument) &
                [t] on_board(instrument, satellite) &
                [t] supports(instrument, mode) &
                [t] power_on(instrument) &
                [t] pointing(satellite, direction) &
                ([t] !have_image_generalized(direction, mode)) &
                !$committed(t+1, have_image_generalized(direction, mode), true)
 :duration      7
 :context
   :effects     [+7] have_image(direction, mode) := true,
                [+1,+7] power_on(instrument) := true,
                [+1,+7] pointing(satellite, direction) := true,
                [+1] have_image_generalized(direction, mode) := true

#operator switch_on(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] power_avail(satellite) &
                [t] !power_on(instrument) &
                [t] !power_on_generalized(instrument)
 :duration      2
 :context
   :effects     [+2] power_on(instrument) := true,
                [+1] calibrated(instrument) := false,
                [+1] power_avail(satellite) := false,
                [+1] power_on_generalized(instrument) := true,
                // Powering on more than one instrument on this satellite
                // at the same time will give sem_power_on conflicting
                // values and is therefore impossible.
                [+1] sem_power_on(satellite, instrument) := true

#operator turn_to(satellite, new_direction, old_direction)
 :iterate (satellite, old_direction, new_direction)
```

```
 :at t
 :precond       [t] pointing(satellite, old_direction) &
                [t] new_direction != old_direction
 :duration      5
 :context
   :effects     [+1] turning_towards(satellite, new_direction) := true,
                [+1] pointing(satellite, old_direction) := false,
                [+5] pointing(satellite, new_direction) := true,
                [+5] turning_towards(satellite, new_direction) := false

#operator switch_off(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] power_on(instrument) &
                [t] power_on_generalized(instrument)
 :duration      1
 :context
   :effects     [+1] power_on(instrument) := false,
                [+1] power_on_generalized(instrument) := false,
                [+1] power_avail(satellite) := true

#operator calibrate(satellite, instrument, direction)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] calibration_target(instrument, direction) &
                [t] pointing(satellite, direction) &
                [t] power_on(instrument) &
                [t] !calibrated(instrument) &
                [t] !calibrating(instrument)
 :duration      5
 :context
   :effects     [+5] calibrated(instrument) := true,
                [+1,+5] power_on(instrument) := true,
                [+1] calibrating(instrument) := true,
                [+5] calibrating(instrument) := false



#control :name "only-take-pictures-of-goals"
         forall t, direction, mode [
                [t] !have_image_generalized(direction, mode) &
                [t+1] have_image_generalized(direction, mode) ->
                goal(have_image(direction, mode)) ]

#control :name "only-point-in-goal-directions"
         forall t, satellite, direction [
                [t] pointing(satellite, direction) ->
                ([t+1] pointing(satellite, direction)) |
                exists new_direction [
                       [t+1] turning_towards(satellite, new_direction) &
                       ([t] goal_direction(satellite, new_direction)) ]]

#define [t] goal_direction(satellite, direction):
        ([t] take_image_possible(satellite, direction)) |
        exists instrument [
                [t] calibration_target(instrument, direction) &
                [t] on_board(instrument, satellite) &
                [t] !calibrated(instrument) &
                [t] power_on(instrument) ] |
        (goal(pointing(satellite, direction)) &
         [t] all_images_collected)

#define [t] take_image_possible(satellite, direction):
        exists mode [
```

```
               goal(have_image(direction, mode)) &
               ([t] !have_image(direction, mode)) &
               exists instrument [
                      [t] supports(instrument, mode) &
                      [t] on_board(instrument, satellite) &
                      [t] power_on(instrument) &
                      [t] calibrated(instrument) ]]

#define [t] all_images_collected:
        !exists direction, mode [
                goal(have_image(direction, mode)) &
                [t] !have_image(direction, mode) ]

#control :name "don't-all-point-in-same-direction"
         forall t, satellite, direction [
                [t] !turning_towards(satellite, direction) ->
                ([t+1] !turning_towards(satellite, direction)) |
                !exists satellite2 [
                        $committed(t+1, turning_towards(satellite2, direction),
true) ] ]

#define [t] usefulness(instrument):
        value(t, $sum(<mode>,
                      [t] supports(instrument, mode) &
                          mode_needed_for_goal(mode),
                      1))

#define [t] mode_needed_for_goal(mode):
        exists direction [
               goal(have_image(direction, mode)) &
               [t] !have_image(direction, mode) ]

#control :name "use-the-most-useful-instrument"
         forall t, instrument [
                [t] !power_on_generalized(instrument) ->
                ([t+1] !power_on_generalized(instrument)) |
                ([t] usefulness(instrument) > 0) &
                !exists instrument2 [
                        [t] usefulness(instrument2) > usefulness(instrument) |
                        [t] power_on(instrument2) ]]

#control :name "don't-switch-instrument-off-if-you-don't-have-to"
         forall t, instrument [
                [t] power_on_generalized(instrument) ->
                ([t+1] power_on_generalized(instrument)) |
                !exists mode [
                        [t] supports(instrument, mode) &
                        [t] mode_needed_for_goal(mode) ]]
```

B.15 Satellite Timed

```
#timescale 0.001

#domain integer :integer :lb 0 :ub 1000000
#domain fixedpoint :fixedpoint :lb 0 :ub 100000 :decimals 4

#domain satellite :elements {}
#domain direction :elements {}
#domain instrument :elements {}
#domain mode :elements {}

#valuevar old_direction, new_direction :domain direction
```

```
#feature on_board(instrument, satellite) :domain boolean :function
#feature supports(instrument, mode) :domain boolean :function
#feature pointing(satellite, direction) :domain boolean :injective
#feature power_avail(satellite) :domain boolean
#feature power_on(instrument) :domain boolean
#feature calibrated(instrument) :domain boolean
#feature have_image(direction, mode) :domain boolean
#feature calibration_target(instrument, direction) :domain boolean :function

#feature slew_time(direction, direction) :domain fixedpoint :function
#feature calibration_time(instrument, direction) :domain fixedpoint :function

#feature turning_towards(satellite, direction) :domain boolean :injective
#feature have_image_generalized(direction, mode) :domain boolean
#feature power_on_generalized(instrument) :domain boolean
#feature calibrating(instrument) :domain boolean
#feature sem_power_on(satellite, instrument) :domain boolean :injective

#deffeature goal_direction(satellite, direction) :domain boolean
#deffeature all_images_collected :domain boolean
#deffeature take_image_possible(satellite, direction) :domain boolean
#deffeature usefulness(instrument) :domain integer
#deffeature mode_needed_for_goal(mode) :domain boolean

#resource sem_take_image(instrument, mode) :domain integer :preference :none

#assert forall t, instrument [
               [t] power_on(instrument) -> power_on_generalized(instrument) ]
#dom [0] forall instrument, mode [
               $init(sem_take_image(instrument, mode)) == 1 &
               $minimum(sem_take_image(instrument, mode)) == 0 &
               $maximum(sem_take_image(instrument, mode)) == 1 ]



#operator take_image(satellite, direction, instrument, mode)
 :at t
 :precond       [t] calibrated(instrument) &
                [t] on_board(instrument, satellite) &
                [t] supports(instrument, mode) &
                [t] power_on(instrument) &
                [t] pointing(satellite, direction) &
                [t] !have_image_generalized(direction, mode)
 :duration      7000 :as t2
 :resources     [+1,+t2] :borrow sem_take_image(instrument, mode) :amount 1
 :context
   :effects     [+t2] have_image(direction, mode) := true,
                [+1,+t2] power_on(instrument) := true,
                [+1,+t2] pointing(satellite, direction) := true,
                [+1] have_image_generalized(direction, mode) := true

#operator switch_on(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                ([t] power_avail(satellite)) &
                ([t] !power_on(instrument)) &
                ([t] !power_on_generalized(instrument))
 :duration      2000 :as t2
 :context
   :effects     [+t2] power_on(instrument) := true,
                [+1] calibrated(instrument) := false,
                [+1] power_avail(satellite) := false,
                [+1] power_on_generalized(instrument) := true,
                [+1] sem_power_on(satellite, instrument) := true
```

```
#operator turn_to(satellite, new_direction, old_direction)
 :iterate (satellite, old_direction, new_direction)
 :at t
 :precond       [t] pointing(satellite, old_direction) &
                [t] new_direction != old_direction
 :duration      $maketime($cast(fixedpoint,
                                integer,
                                value(t, 1000.0 * slew_time(old_direction,
new_direction)))) :as t2
 :context
   :effects     [+1] turning_towards(satellite, new_direction) := true,
                [+1] pointing(satellite, old_direction) := false,
                [+t2] pointing(satellite, new_direction) := true,
                [+t2] turning_towards(satellite, new_direction) := false

#operator switch_off(instrument, satellite)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] power_on(instrument) &
                [t] power_on_generalized(instrument)
 :duration      1000 :as t2
 :context
   :effects     [+t2] power_on(instrument) := false,
                [+1] power_on_generalized(instrument) := false,
                [+t2] power_avail(satellite) := true

#operator calibrate(satellite, instrument, direction)
 :at t
 :precond       [t] on_board(instrument, satellite) &
                [t] calibration_target(instrument, direction) &
                [t] pointing(satellite, direction) &
                [t] power_on(instrument) &
                [t] !calibrated(instrument) &
                [t] !calibrating(instrument)
 :duration      $maketime($cast(fixedpoint,
                                integer,
                                value(t, 1000.0 * calibration_time(instrument,
direction)))) :as t2
 :context
   :effects     [+t2] calibrated(instrument) := true,
                [+1,+t2] power_on(instrument) := true,
                [+1] calibrating(instrument) := true,
                [+t2] calibrating(instrument) := false



#control :name "only-take-pictures-of-goals2"
         forall t, direction, mode [
                [t] !have_image_generalized(direction, mode) &
                [t+1] have_image_generalized(direction, mode) ->
                goal(have_image(direction, mode)) ]

#control :name "only-point-in-goal-directions"
         forall t, satellite, direction [
                [t] pointing(satellite, direction) ->
                ([t+1] pointing(satellite, direction)) |
                exists new_direction [
                       [t+1] turning_towards(satellite, new_direction) &
                       ([t] goal_direction(satellite, new_direction)) ]]

#define [t] goal_direction(satellite, direction):
        ([t] take_image_possible(satellite, direction)) |
        exists instrument [
```

```
                [t] calibration_target(instrument, direction) &
                [t] on_board(instrument, satellite) &
                [t] !calibrated(instrument) &
                [t] power_on(instrument) ] |
        (goal(pointing(satellite, direction)) &
         [t] all_images_collected)

#define [t] take_image_possible(satellite, direction):
        exists mode [
               goal(have_image(direction, mode)) &
               ([t] !have_image(direction, mode)) &
               exists instrument [
                      [t] supports(instrument, mode) &
                      [t] on_board(instrument, satellite) &
                      [t] power_on(instrument) &
                      [t] calibrated(instrument) ]]

#define [t] all_images_collected:
        !exists direction, mode [
                goal(have_image(direction, mode)) &
                [t] !have_image(direction, mode) ]

#control :name "don't-all-point-in-same-direction"
         forall t, satellite, direction [
                [t] !turning_towards(satellite, direction) ->
                ([t+1] !turning_towards(satellite, direction)) |
                !exists satellite2 [
                        $committed(t+1,
                                   turning_towards(satellite2, direction),
                                   true) |
                        [t] turning_towards(satellite2, direction) ] ]

#define [t] usefulness(instrument):
        value(t, $sum(<mode>,
                      [t] supports(instrument, mode) &
                          mode_needed_for_goal(mode),
                      1))

#define [t] mode_needed_for_goal(mode):
        exists direction [
               goal(have_image(direction, mode)) &
               [t] !have_image(direction, mode) ]

#control :name "use-the-most-useful-instrument"
         forall t, instrument [
                [t] !power_on_generalized(instrument) ->
                ([t+1] !power_on_generalized(instrument)) |
                ([t] usefulness(instrument) > 0) &
                !exists instrument2 [
                        [t] usefulness(instrument2) > usefulness(instrument) |
                        [t] power_on(instrument2) ]]

#control :name "don't-switch-instrument-off-if-you-don't-have-to"
         forall t, instrument [
                [t] power_on_generalized(instrument) ->
                ([t+1] power_on_generalized(instrument)) |
                !exists mode [
                        [t] supports(instrument, mode) &
                        [t] mode_needed_for_goal(mode) ]]
```

B.16 UMTranslog-2

```
#domain integer :integer :lb 0 :ub 10000
```

```
#domain object :elements { regularp, bulky, liquid, granular, cars, mail,
                           regularv, flatbed, tanker, hopper, auto, air,
                           truck, airplane, train,
                           road-route, rail-route, air-route,
                           airport, train-station }
#domain region :parent object :elements {}
#domain city :parent object :elements {}
#domain location :parent object :elements {}
#domain package :parent object :elements {}
#domain vehicle :parent object :elements {}
#domain route :parent object :elements {}
#domain equipment :parent object :elements {}
// The package types.
#domain ptype :parent object :elements { regularp, bulky, liquid,
                                         granular, cars, mail }
// The vehicle subtypes.
#domain vtype :parent object :elements { regularv, flatbed, tanker,
                                         hopper, auto, air }
// The vehicle types.
#domain vptype :parent object :elements { truck, airplane, train }
// The route types.
#domain rtype :parent object :elements { road-route, rail-route, air-route }
// The location types.
#domain ltype :parent object :elements { airport, train-station }
#domain crane :parent equipment :elements {}
#domain plane-ramp :parent equipment :elements {}

#valuevar location-from, location-to, location-goal :domain location
#valuevar city-from, city-to :domain city

#feature at-equipment(equipment, location) :domain boolean :function :injective
#feature at-packagec(package, crane) :domain boolean :injective
#feature at-packagel(package, location) :domain boolean :injective
#feature at-packagev(package, vehicle) :domain boolean :injective
#feature at-vehicle(vehicle, location) :domain boolean :injective
#feature availablel(location) :domain boolean :function
#feature availabler(route) :domain boolean :function
#feature availablev(vehicle) :domain boolean :function
#feature chute-connected(vehicle) :domain boolean
#feature clear :domain boolean
#feature connect-city(route, rtype, city1, city2) :domain boolean :function
#feature connect-loc(route, rtype, location1, location2)
         :domain boolean :function
#feature delivered(package, location) :domain boolean :injective
#feature door-open(vehicle) :domain boolean
#feature empty(crane) :domain boolean
#feature fees-collected(package) :domain boolean
#feature hose-connected(vehicle) :domain boolean
#feature h-start(package) :domain boolean
#feature hub(location) :domain boolean :function
#feature in-city(location, city) :domain boolean :function
#feature in-region(city, region) :domain boolean :function
#feature move(package) :domain boolean
#feature move-emp(vehicle) :domain boolean
#feature over(package) :domain boolean
#feature pv-compatible(ptype, vtype) :domain boolean :function
#feature ramp-connected(vehicle, plane-ramp) :domain boolean :injective
#feature ramp-down(vehicle) :domain boolean
#feature rv-compatible(rtype, vptype) :domain boolean :function
#feature serves(location, region) :domain boolean :function
#feature tcenter(location) :domain boolean :function
#feature t-end(package) :domain boolean
#feature t-start(package) :domain boolean
```

```
#feature typel(location, ltype) :domain boolean :function :injective
#feature typep(package, ptype) :domain boolean :function :injective-always
// Vehicle is of subtype vtype.
#feature typev(vehicle, vtype) :domain boolean :function :injective-always
// Vehicle ís of type vptype.
#feature typevp(vehicle, vptype) :domain boolean :function :injective-always
#feature unload(vehicle) :domain boolean
#feature valve-open(vehicle) :domain boolean

#feature distance(location1, location2) :domain integer :function
#feature gas-left(vehicle) :domain integer
#feature gpm(vehicle) :domain integer :function
#feature height-v(vehicle) :domain integer :function
#feature height-cap-l(location) :domain integer :function
#feature height-cap-r(route) :domain integer :function
#feature length-v(vehicle) :domain integer :function
#feature length-cap-l(location) :domain integer :function
#feature local-height(city) :domain integer :function
#feature local-weight(city) :domain integer :function
#feature volume-cap-c(crane) :domain integer :function
#feature volume-cap-l(location) :domain integer :function
#feature volume-cap-v(vehicle) :domain integer :function
#feature volume-load-l(location) :domain integer :function
#feature volume-load-v(vehicle) :domain integer :function
#feature volume-p(package) :domain integer :function
#feature weight-cap-c(crane) :domain integer :function
#feature weight-cap-r(route) :domain integer :function
#feature weight-cap-v(vehicle) :domain integer :function
#feature weight-p(package) :domain integer :function
#feature weight-load-v(vehicle) :domain integer :function
#feature weight-v(vehicle) :domain integer :function
#feature width-v(vehicle) :domain integer :function
#feature width-cap-l(location) :domain integer :function

#deffeature in-wrong-city(package, location) :domain boolean
#deffeature in-same-city(location1, location2) :domain boolean
#deffeature at-packagel-generalized(package, location) :domain boolean

#deffeature package-vehicle-compatible(package, vehicle) :domain boolean
#deffeature need-to-move-package-from(package, location) :domain boolean
#deffeature need-to-unload-package-at(package, location) :domain boolean

#deffeature reasonable-vehicle-for-package(package, vehicle, location)
            :domain boolean :uncached
#deffeature reasonable-nontruck-for-package(package, vehicle, location)
            :domain boolean :uncached
#deffeature reasonable-truck-for-package(package, vehicle, location)
            :domain boolean :uncached
#deffeature reasonable-truck-location(vehicle, location, location)
            :domain boolean :uncached
#deffeature can-go-by-truck(vehicle, location, location)
            :domain boolean :uncached
#deffeature reasonable-nontruck-location(vehicle, location, location)
            :domain boolean :uncached
#deffeature can-go-by-nontruck(vehicle, location, location)
            :domain boolean :uncached

#resource rvolume-load-l(location) :domain integer :preference :none
#resource rvolume-load-v(vehicle) :domain integer :preference :none
#resource rweight-load-v(vehicle) :domain integer :preference :none
#resource rgas-left(vehicle) :domain integer :preference :none

#resource rpackages-to-deliver :domain integer :preference :none
```

```
#dom [0] forall location [
               $init(rvolume-load-l(location)) == volume-load-l(location) &
               $minimum(rvolume-load-l(location)) == 0 &
               $maximum(rvolume-load-l(location)) == volume-cap-l(location) ]
#dom [0] forall vehicle [
               $init(rvolume-load-v(vehicle)) == volume-load-v(vehicle) &
               $minimum(rvolume-load-v(vehicle)) == 0 &
               $maximum(rvolume-load-v(vehicle)) == volume-cap-v(vehicle) ]
#dom [0] forall vehicle [
               $init(rweight-load-v(vehicle)) == weight-load-v(vehicle) &
               $minimum(rweight-load-v(vehicle)) == 0 &
               $maximum(rweight-load-v(vehicle)) == weight-cap-v(vehicle) ]
#dom [0] forall vehicle [
               $init(rgas-left(vehicle)) == gas-left(vehicle) &
               $minimum(rgas-left(vehicle)) == 0 &
               $maximum(rgas-left(vehicle)) == 9999 ]

#dom [0] $init(rpackages-to-deliver) ==
         $sum(<package>,
              exists location [
                     goal(delivered(package, location)) ],
              1) &
         $minimum(rpackages-to-deliver) == 0 &
         $maximum(rpackages-to-deliver) == 9999

#assert forall t, package, location, vehicle [
               [t] at-packagev(package, vehicle) ->
                   !at-packagel(package, location) ]
#assert forall t, package, location, crane [
               [t] at-packagec(package, crane) ->
                   !at-packagel(package, location) ]
#assert forall t, package, vehicle, crane [
               [t] at-packagec(package, crane) ->
                   !at-packagev(package, vehicle) ]
#assert forall t, package, location, location2 [
               [t] delivered(package, location) ->
                   !at-packagel(package, location2) ]



// When all packages have been delivered, the planner must make sure that all
// vehicle doors and valves are closed and all loading equipment disconnected.
#operator clean-domain
 :at t
 :precond       !exists vehicle [
                        [t] unload(vehicle) &
                        (([t] typev(vehicle, regularv) &
                          [t] door-open(vehicle)) |
                         ([t] typev(vehicle, hopper) &
                          [t] chute-connected(vehicle)) |
                         ([t] typev(vehicle, tanker) &
                          [t] hose-connected(vehicle)) |
                         ([t] typev(vehicle, tanker) &
                          [t] valve-open(vehicle)) |
                         ([t] typev(vehicle, auto) &
                          [t] ramp-down(vehicle)) |
                         (([t] typev(vehicle, air)) &
                          exists plane-ramp [
                                 [t] ramp-connected(vehicle, plane-ramp) ]) |
                         ([t] typev(vehicle, air) &
                          [t] door-open(vehicle))) ] &
                [t] $available(rpackages-to-deliver) == 0
 :context
   :effects     [+1] clear := true
```

```
#operator collect-fees(package)
 :at t
 :precond       ([t] !fees-collected(package)) &
                !exists location [
                        [t] delivered(package, location) ]
 :context
   :effects     [+1] fees-collected(package) := true

#operator deliver(package, location)
 :at t
 :precond       ([t] at-packagel(package, location)) &
                !exists location2 [
                        [t] delivered(package, location2) ]
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :consume rpackages-to-deliver :amount 1
 :context
   :effects     [+1] delivered(package, location) := true,
                [+1] at-packagel(package, location) := false

// Regular vehicles must open the door before loading or unloading packages.
#operator open-door-regular(vehicle)
 :at t
 :precond       [t] !door-open(vehicle) &
                [t] typev(vehicle, regularv)
 :context
   :effects     [+1] door-open(vehicle) := true

// And close the door before going anywhere.
#operator close-door-regular(vehicle)
 :at t
 :precond       [t] door-open(vehicle) &
                [t] typev(vehicle, regularv)
 :context
   :effects     [+1] door-open(vehicle) := false

// Load a package into a regular vehicle.
#operator load-regular(package, vehicle, location)
 :at t
 :precond       [t] at-vehicle(vehicle, location) &
                [t] availablev(vehicle) &
                [t] at-packagel(package, location) &
                ([t] typev(vehicle, regularv)) &
                exists ptype [
                       [t] typep(package, ptype) &
                       [t] pv-compatible(ptype, regularv) ] &
                [t] door-open(vehicle) &
                [t] fees-collected(package)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagel(package, location) := false

// Unload a package from a regular vehicle.
#operator unload-regular(package, vehicle, location)
 :at t
 :precond       [t] at-vehicle(vehicle, location) &
                [t] at-packagev(package, vehicle) &
                [t] typev(vehicle, regularv) &
                [t] door-open(vehicle)
```

```
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-
p(package),
                [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] at-packagev(package, vehicle) := false,
                [+1] move(package) := false,
                [+1] unload(vehicle) := true,
                [+1] clear := false

// Use a crane to pick up a package.
#operator pick-up-package-ground(package, crane, location)
 :at t
 :precond       [t] at-equipment(crane, location) &
                [t] at-packagel(package, location) &
                [t] empty(crane) &
                [t] fees-collected(package) &
                [t] weight-p(package) <= weight-cap-c(crane) &
                [t] volume-p(package) <= volume-cap-c(crane)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-p(package)
 :context
   :effects     [+1] at-packagec(package, crane) := true,
                [+1] empty(crane) := false,
                [+1] at-packagel(package, location) := false

// Use a crane to load a package on a flatbed vehicle.
#operator put-down-package-vehicle(package, crane, vehicle, location)
 :at t
 :precond       [t] at-equipment(crane, location) &
                [t] at-packagec(package, crane) &
                [t] at-vehicle(vehicle, location) &
                [t] typev(vehicle, flatbed) &
                ([t] availablev(vehicle)) &
                exists ptype [
                       [t] typep(package, ptype) &
                       [t] pv-compatible(ptype, flatbed) ] &
                [t] fees-collected(package)
 :resources     [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] empty(crane) := true,
                [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagec(package, crane) := false

// Use a crane to unload a package from a flatbed vehicle.
#operator pick-up-package-vehicle(package, crane, vehicle, location)
 :at t
 :precond       [t] empty(crane) &
                [t] at-equipment(crane, location) &
                [t] at-packagev(package, vehicle) &
                [t] at-vehicle(vehicle, location) &
                [t] typev(vehicle, flatbed)
 :resources     [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagec(package, crane) := true,
                [+1] empty(crane) := false,
                [+1] at-packagev(package, vehicle) := false

// The crane puts the package on the ground.
#operator put-down-package-ground(package, crane, location)
 :at t
 :precond       [t] at-equipment(crane, location) &
```

```
                [t] at-packagec(package, crane)
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] empty(crane) := true,
                [+1] move(package) := false,
                [+1] at-packagec(package, crane) := false

// Hoppers need to connect a chute before loading or unloading packages.
#operator connect-chute(vehicle)
 :at t
 :precond       [t] !chute-connected(vehicle) &
                [t] typev(vehicle, hopper)
 :context
   :effects     [+1] chute-connected(vehicle) := true

// And disconnect the chute before going anywhere.
#operator disconnect-chute(vehicle)
 :at t
 :precond       [t] chute-connected(vehicle) &
                [t] typev(vehicle, hopper)
 :context
   :effects     [+1] chute-connected(vehicle) := false

// Load a package into a hopper.
#operator fill-hopper(package, vehicle, location)
 :at t
 :precond       [t] chute-connected(vehicle) &
                [t] at-vehicle(vehicle, location) &
                [t] at-packagel(package, location) &
                [t] availablev(vehicle) &
                ([t] typev(vehicle, hopper)) &
                exists ptype [
                       [t] typep(package, ptype) &
                       [t] pv-compatible(ptype, hopper) ] &
                [t] fees-collected(package)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagel(package, location) := false

// Unload a package from a hopper.
#operator empty-hopper(package, vehicle, location)
 :at t
 :precond       [t] chute-connected(vehicle) &
                [t] at-vehicle(vehicle, location) &
                [t] at-packagev(package, vehicle) &
                [t] typev(vehicle, hopper)
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-
p(package),
                [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] at-packagev(package, vehicle) := false,
                [+1] move(package) := false,
                [+1] unload(vehicle) := true,
                [+1] clear := false

// Tankers need to connect a hose before opening the valve.
#operator connect-hose(vehicle)
```

```
 :at t
 :precond       [t] !hose-connected(vehicle) &
                [t] typev(vehicle, tanker)
 :context
   :effects     [+1] hose-connected(vehicle) := true

// And disconnect the hose before going anywhere.
#operator disconnect-hose(vehicle)
 :at t
 :precond       [t] hose-connected(vehicle) &
                [t] !valve-open(vehicle) &
                [t] typev(vehicle, tanker)
 :context
   :effects     [+1] hose-connected(vehicle) := false

// Tankers need to open the valve before loading or unloading packages.
#operator open-valve(vehicle)
 :at t
 :precond       [t] !valve-open(vehicle) &
                [t] hose-connected(vehicle) &
                [t] typev(vehicle, tanker)
 :context
   :effects     [+1] valve-open(vehicle) := true

// And close the valve before disconnecting the hose.
#operator close-valve(vehicle)
 :at t
 :precond       [t] valve-open(vehicle) &
                [t] typev(vehicle, tanker)
 :context
   :effects     [+1] valve-open(vehicle) := false

// Load a package into a tanker.
#operator fill-tank(package, vehicle, location)
 :at t
 :precond       [t] at-vehicle(vehicle, location) &
                [t] at-packagel(package, location) &
                [t] typev(vehicle, tanker) &
                ([t] availablev(vehicle)) &
                exists ptype [
                       [t] typep(package, ptype) &
                       [t] pv-compatible(ptype, tanker) ] &
                [t] valve-open(vehicle) &
                [t] hose-connected(vehicle) &
                [t] fees-collected(package)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagel(package, location) := false

// Unload a package from a tanker.
#operator empty-tank(package, vehicle, location)
 :at t
 :precond       [t] at-vehicle(vehicle, location) &
                [t] at-packagev(package, vehicle) &
                [t] typev(vehicle, tanker) &
                [t] availablev(vehicle) &
                [t] hose-connected(vehicle) &
                [t] valve-open(vehicle)
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-
p(package),
```

```
                [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] at-packagev(package, vehicle) := false,
                [+1] move(package) := false,
                [+1] unload(vehicle) := true,
                [+1] clear := false

// Car transports need to lower the ramp before loading or unloading cars.
#operator lower-ramp(vehicle)
 :at t
 :precond       [t] !ramp-down(vehicle) &
                [t] typev(vehicle, auto)
 :context
   :effects     [+1] ramp-down(vehicle) := true

// And raise the ramp before going anywhere.
#operator raise-ramp(vehicle)
 :at t
 :precond       [t] ramp-down(vehicle) &
                [t] typev(vehicle, auto)
 :context
   :effects     [+1] ramp-down(vehicle) := false

// Load a package into a car transport.
#operator load-cars(package, vehicle, location)
 :at t
 :precond       ([t] typev(vehicle, auto)) &
                exists ptype [
                       [t] typep(package, ptype) &
                       [t] pv-compatible(ptype, auto) ] &
                [t] availablev(vehicle) &
                [t] at-packagel(package, location) &
                [t] at-vehicle(vehicle, location) &
                [t] ramp-down(vehicle) &
                [t] fees-collected(package)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagel(package, location) := false

// Unload a package from a car transport.
#operator unload-cars(package, vehicle, location)
 :at t
 :precond       [t] at-packagev(package, vehicle) &
                [t] at-vehicle(vehicle, location) &
                [t] typev(vehicle, auto) &
                [t] ramp-down(vehicle)
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-
p(package),
                [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] at-packagev(package, vehicle) := false,
                [+1] move(package) := false,
                [+1] unload(vehicle) := true,
                [+1] clear := false

// Aircraft need to attach a conveyor ramp before opening the door.
```

```
#operator attach-conveyor-ramp(vehicle, plane-ramp, location)
 :at t
 :precond       !exists vehicle2 [
                        [t] ramp-connected(vehicle2, plane-ramp) ] &
                [t] at-equipment(plane-ramp, location) &
                ([t] typev(vehicle, air)) &
                !exists plane-ramp2 [
                        [t] ramp-connected(vehicle, plane-ramp2) ] &
                [t] at-vehicle(vehicle, location)
 :context
   :effects     [+1] ramp-connected(vehicle, plane-ramp) := true

// And detach the ramp before going anywhere.
#operator detach-conveyor-ramp(vehicle, plane-ramp, location)
 :at t
 :precond       [t] ramp-connected(vehicle, plane-ramp) &
                [t] at-equipment(plane-ramp, location) &
                [t] at-vehicle(vehicle, location) &
                [t] !door-open(vehicle)
 :context
   :effects     [+1] ramp-connected(vehicle, plane-ramp) := false

// Aircraft need to open the door before loading or unloading packages.
#operator open-door-airplane(vehicle)
 :at t
 :precond       [t] !door-open(vehicle) &
                ([t] typev(vehicle, air)) &
                exists plane-ramp [
                        [t] ramp-connected(vehicle, plane-ramp) ]
 :context
   :effects     [+1] door-open(vehicle) := true

// Load package into aircraft.
#operator load-airplane(package, vehicle, location)
 :at t
 :precond       [t] at-packagel(package, location) &
                [t] at-vehicle(vehicle, location) &
                ([t] availablev(vehicle)) &
                exists ptype [
                        [t] typep(package, ptype) &
                        [t] pv-compatible(ptype, air) ] &
                ([t] door-open(vehicle)) &
                exists plane-ramp [
                        [t] ramp-connected(vehicle, plane-ramp) ] &
                [t] fees-collected(package)
 :resources     [+1] :consume rvolume-load-l(location) :amount volume-
p(package),
                [+1] :produce rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :produce rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagev(package, vehicle) := true,
                [+1] at-packagel(package, location) := false

// Unload package from aircraft.
#operator unload-airplane(package, vehicle, location)
 :at t
 :precond       [t] typev(vehicle, air) &
                [t] at-packagev(package, vehicle) &
                ([t] at-vehicle(vehicle, location)) &
                exists plane-ramp [
                        [t] ramp-connected(vehicle, plane-ramp) ] &
                [t] door-open(vehicle)
 :resources     [+1] :produce rvolume-load-l(location) :amount volume-
p(package),
```

```
                [+1] :consume rweight-load-v(vehicle) :amount weight-p(package),
                [+1] :consume rvolume-load-v(vehicle) :amount volume-p(package)
 :context
   :effects     [+1] at-packagel(package, location) := true,
                [+1] at-packagev(package, vehicle) := false,
                [+1] move(package) := false,
                [+1] unload(vehicle) := true,
                [+1] clear := false

// And close the door before detaching the ramp.
#operator close-door-airplane(vehicle)
 :at t
 :precond       [t] door-open(vehicle) &
                [t] typev(vehicle, air)
 :context
   :effects     [+1] door-open(vehicle) := false

// Move a truck between two locations in the same city.
// Either both or none of the locations are transportation centers.
// All packages in the truck will be over(package) and not allowed to move
// again.
#operator move-vehicle-local-road-route1(vehicle, location-from, location-to,
city)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] typevp(vehicle, truck) &
                [t] in-city(location-from, city) &
                [t] in-city(location-to, city) &
                [t] height-v(vehicle) <= local-height(city) &
                [t] weight-v(vehicle) + weight-load-v(vehicle) <=
                    local-weight(city) &
                (([t] tcenter(location-from) &
                  [t] tcenter(location-to)) |
                 ([t] !tcenter(location-from) &
                  [t] !tcenter(location-to))) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-start(package) |
                         [t] t-end(package) |
                         [t] h-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
```

```
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] move(package) := true,
                [+1] over(package) := true

// Move a truck between two locations in the same city.
// The trip is from a non transportation center to a transportation center.
// All packages in the truck will be t-start(package) and further transportation
// must be by train or plane.
#operator move-vehicle-local-road-route2(vehicle, location-from, location-to,
city)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] typevp(vehicle, truck) &
                [t] in-city(location-from, city) &
                [t] in-city(location-to, city) &
                [t] height-v(vehicle) <= local-height(city) &
                [t] weight-v(vehicle) + weight-load-v(vehicle) <=
                    local-weight(city) &
                [t] !tcenter(location-from) &
                ([t] tcenter(location-to)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-start(package) |
                         [t] t-end(package) |
                         [t] h-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
```

```
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] move(package) := true,
                [+1] t-start(package) := true

// Move a truck between two locations in the same city.
// The trip is from a transportation center to a non transportation center.
// All packages in the truck will be over(package) and not allowed to move
// again.
#operator move-vehicle-local-road-route3(vehicle, location-from, location-to,
city)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] typevp(vehicle, truck) &
                [t] in-city(location-from, city) &
                [t] in-city(location-to, city) &
                [t] height-v(vehicle) <= local-height(city) &
                [t] weight-v(vehicle) + weight-load-v(vehicle) <=
                    local-weight(city) &
                [t] tcenter(location-from) &
                ([t] !tcenter(location-to)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] over(package) := true,
```

```
                [+1] move(package) := true,
                [+1] t-end(package) := false,
                [+1] h-start(package) := false

// Move a truck between two locations in different cities using a road route.
// All packages in the truck will be over(package) and not allowed to move
// again.
#operator move-vehicle-road-route-crossCity(vehicle, location-from, location-to,
city-from, city-to, route)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] typevp(vehicle, truck) &
                [t] in-city(location-from, city-from) &
                [t] in-city(location-to, city-to) &
                [t] city-from != city-to &
                [t] connect-city(route, road-route, city-from, city-to) &
                [t] availabler(route) &
                [t] height-v(vehicle) <= height-cap-r(route) &
                ([t] weight-v(vehicle) + weight-load-v(vehicle) <=
                     weight-cap-r(route)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-start(package) |
                         [t] t-end(package) |
                         [t] h-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] move(package) := true,
                [+1] over(package) := true

// Move a train or plane between two locations using a compatible route.
```

```
// Either both or none of the locations are transportation hubs.
// All packages in the truck will be t-end(package) and only allowed to be moved
// by truck to a non transportation center.
#operator move-vehicle-nonroad-route1(vehicle, location-from, location-to,
route)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                  [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] !typevp(vehicle, truck) &
                [t] tcenter(location-from) &
                [t] tcenter(location-to) &
                (([t] hub(location-from) &
                  [t] hub(location-to)) |
                 ([t] !hub(location-from) &
                  [t] !hub(location-to))) &
                [t] availablel(location-from) &
                ([t] availablel(location-to)) &
                exists rtype, vptype [
                       [t] connect-loc(route,
                                       rtype,
                                       location-from,
                                       location-to) &
                       [t] typevp(vehicle, vptype) &
                       [t] rv-compatible(rtype, vptype) ] &
                [t] availabler(route) &
                [t] height-v(vehicle) <= height-cap-r(route) &
                ([t] weight-v(vehicle) + weight-load-v(vehicle) <=
                     weight-cap-r(route)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-end(package) |
                         [t] h-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
```

```
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] t-end(package) := true,
                [+1] t-start(package) := false,
                [+1] move(package) := true

// Move a train or plane between two locations using a compatible route.
// The trip is from a non transportation hub to a transportation hub.
// All packages in the truck will be h-start(package) and only allowed to be
moved
// by truck to a non transportation center or by train or plane to a non
// transportation hub.
#operator move-vehicle-nonroad-route2(vehicle, location-from, location-to,
route)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] !typevp(vehicle, truck) &
                [t] tcenter(location-from) &
                [t] tcenter(location-to) &
                [t] !hub(location-from) &
                [t] hub(location-to) &
                [t] availablel(location-from) &
                ([t] availablel(location-to)) &
                exists rtype, vptype [
                       [t] connect-loc(route,
                                       rtype,
                                       location-from,
                                       location-to) &
                       [t] typevp(vehicle, vptype) &
                       [t] rv-compatible(rtype, vptype) ] &
                [t] availabler(route) &
                [t] height-v(vehicle) <= height-cap-r(route) &
                ([t] weight-v(vehicle) + weight-load-v(vehicle) <=
                     weight-cap-r(route)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-end(package) |
                         [t] h-start(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
                     :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
```

```
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] h-start(package) := true,
                [+1] t-start(package) := false,
                [+1] move(package) := true

// Move a train or plane between two locations using a compatible route.
// The trip is from a transportation hub to a non transportation hub.
// All packages in the truck will be t-end(package) and only allowed to be moved
// by truck to a non transportation center.
#operator move-vehicle-nonroad-route3(vehicle, location-from, location-to,
route)
 :at t
 :precond       [t] at-vehicle(vehicle, location-from) &
                [t] location-from != location-to &
                ([t] $available(rvolume-load-v(vehicle)) > 0 |
                 [t] !move-emp(vehicle)) &
                (([t] typev(vehicle, regularv) &
                  [t] !door-open(vehicle)) |
                 ([t] typev(vehicle, hopper) &
                  [t] !chute-connected(vehicle)) |
                 ([t] typev(vehicle, tanker) &
                  [t] !hose-connected(vehicle)) |
                 ([t] typev(vehicle, auto) &
                  [t] !ramp-down(vehicle)) |
                 ([t] typev(vehicle, air) &
                  ([t] !door-open(vehicle)) &
                  !exists plane-ramp [
                          [t] ramp-connected(vehicle, plane-ramp) ]) |
                 [t] typev(vehicle, flatbed)) &
                [t] height-cap-l(location-to) >= height-v(vehicle) &
                [t] length-cap-l(location-to) >= length-v(vehicle) &
                [t] width-cap-l(location-to) >= width-v(vehicle) &
                [t] !typevp(vehicle, truck) &
                [t] tcenter(location-from) &
                [t] tcenter(location-to) &
                [t] hub(location-from) &
                [t] !hub(location-to) &
                [t] availablel(location-from) &
                ([t] availablel(location-to)) &
                exists rtype, vptype [
                       [t] connect-loc(route,
                                       rtype,
                                       location-from,
                                       location-to) &
                       [t] typevp(vehicle, vptype) &
                       [t] rv-compatible(rtype, vptype) ] &
                [t] availabler(route) &
                [t] height-v(vehicle) <= height-cap-r(route) &
                ([t] weight-v(vehicle) + weight-load-v(vehicle) <=
                     weight-cap-r(route)) &
                !exists package [
                        [t] at-packagev(package, vehicle) &
                        ([t] over(package) |
                         [t] move(package) |
                         [t] t-end(package)) ]
 :resources     [+1] :consume rgas-left(vehicle)
```

```
                    :amount gpm(vehicle) * distance(location-from, location-to)
 :context
   :effects     [+1] at-vehicle(vehicle, location-from) := false,
                [+1] at-vehicle(vehicle, location-to) := true
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) > 0
   :effects     [+1] move-emp(vehicle) := false
 :context
   :precond     [t] $available(rvolume-load-v(vehicle)) == 0
   :effects     [+1] move-emp(vehicle) := true
 :context
   :forall      package
   :precond     [t] at-packagev(package, vehicle)
   :effects     [+1] t-end(package) := true,
                [+1] h-start(package) := false,
                [+1] t-start(package) := false,
                [+1] move(package) := true



// ------------------------ Movement trains and planes -----------------------

// Trains and planes only go to locations that are reasonable-nontruck-location.
#control :name "only-move-nontrucks-to-reasonable-locations"
         forall t, vehicle, location-from [
                [t] !typevp(vehicle, truck) &
                [t] at-vehicle(vehicle, location-from) ->
                ([t+1] at-vehicle(vehicle, location-from)) |
                exists location-to [
                       [t+1] at-vehicle(vehicle, location-to) &
                       [t] reasonable-nontruck-location(vehicle,
                                                        location-from,
                                                        location-to) ] ]

// A location is reasonable for a train or plane if:
#define [t] reasonable-nontruck-location(vehicle, location-from, location-to):
        // There's a package to pick up.
        exists package [
               [t] at-packagel-generalized(package, location-to) &
               [t] !over(package) ] |
        // We're carrying a package that needs to go there.
        exists package [
               [t] at-packagev(package, vehicle) &
               goal(delivered(package, location-to)) ] |
        // We're carrying a package going to that city.
        exists package, location-goal [
               [t] at-packagev(package, vehicle) &
               goal(delivered(package, location-goal)) &
               [t] in-same-city(location-goal, location-to) &
               [t] !can-go-by-nontruck(vehicle,
                                       location-from,
                                       location-goal) ] |
        // We're carrying a package going to a city we can't reach so we need
        // to go through an intermediate city.
        exists package, location-goal [
               [t] at-packagev(package, vehicle) &
               goal(delivered(package, location-goal)) &
               ([t] !in-same-city(location-goal, location-to)) &
               [t] !hub(location-from) &
               [t] hub(location-to) &
               ([t] !can-go-by-nontruck(vehicle,
                                        location-from,
                                        location-goal)) &
               !exists location3 [
```

```
                        [t] in-same-city(location3, location-goal) &
                        [t] can-go-by-nontruck(vehicle,
                                               location-from,
                                               location3) ] ]

// A train or plane can travel between location-from and location-to if there
// is enough gas, the vehicle meets all size restrictions, both locations are
// available transportation centers and there is an available and compatible
// route to travel by.
#define [t] can-go-by-nontruck(vehicle, location-from, location-to):
        [t] $available(rgas-left(vehicle)) >=
            distance(location-from, location-to) * gpm(vehicle) &
        [t] height-cap-l(location-to) >= height-v(vehicle) &
        [t] length-cap-l(location-to) >= length-v(vehicle) &
        [t] width-cap-l(location-to) >= width-v(vehicle) &
        [t] tcenter(location-from) &
        [t] tcenter(location-to) &
        [t] availablel(location-from) &
        ([t] availablel(location-to)) &
        exists route, rtype, vptype [
               [t] connect-loc(route, rtype, location-from, location-to) &
               [t] typevp(vehicle, vptype) &
               [t] rv-compatible(rtype, vptype) &
               [t] availabler(route) &
               [t] height-v(vehicle) <= height-cap-r(route) &
               ([t] weight-v(vehicle) + $available(rweight-load-v(vehicle)) <=
                    weight-cap-r(route)) ]

// ----------------------------- Movement trucks -----------------------------

// Trucks only go to locations that are reasonable-truck-location.
#control :name "only-move-trucks-to-reasonable-locations"
         forall t, vehicle, location-from [
                [t] typevp(vehicle, truck) &
                [t] at-vehicle(vehicle, location-from) ->
                ([t+1] at-vehicle(vehicle, location-from)) |
                exists location-to [
                       [t+1] at-vehicle(vehicle, location-to) &
                       [t] reasonable-truck-location(vehicle,
                                                     location-from,
                                                     location-to) ] ]

// A location is reasonable for a truck if:
#define [t] reasonable-truck-location(vehicle, location-from, location-to):
        // There's a package to pick up and we're not carrying a package that
        // needs to go elsewhere (since that package may then be impossible
        // to deliver later).
        exists package [
               [t] at-packagel-generalized(package, location-to) &
               [t] !over(package) ] &
        !exists package [
                [t] at-packagev(package, vehicle) ] |
        // We're carrying a package that needs to go there.
        exists package [
               [t] at-packagev(package, vehicle) &
               goal(delivered(package, location-to)) ] |
        // We're carrying a package going to another city with no road_route
        // and the location is a tcenter.
        exists package, location-goal [
               [t] at-packagev(package, vehicle) &
               [t] in-wrong-city(package, location-from) &
               [t] in-same-city(location-from, location-to) &
               ([t] tcenter(location-to)) &
               goal(delivered(package, location-goal)) &
```

```
              [t] !can-go-by-truck(vehicle, location-from, location-goal) ]

// A truck can travel between location-from and location-to if there
// is enough gas, the vehicle meets all size restrictions and both locations
// are in the same city or there is an available intercity road route to use.
#define [t] can-go-by-truck(vehicle, location-from, location-to):
        [t] $available(rgas-left(vehicle)) >=
            distance(location-from, location-to) * gpm(vehicle) &
        [t] height-cap-l(location-to) >= height-v(vehicle) &
        [t] length-cap-l(location-to) >= length-v(vehicle) &
        ([t] width-cap-l(location-to) >= width-v(vehicle)) &
        (exists city [
                [t] in-city(location-from, city) &
                [t] in-city(location-to, city) ] |
         exists city-from, city-to [
                [t] in-city(location-from, city-from) &
                ([t] in-city(location-to, city-to)) &
                exists route [
                       [t] connect-city(route, road-route, city-from, city-to) &
                       [t] availabler(route) &
                       [t] height-v(vehicle) <= height-cap-r(route) &
                       [t] weight-v(vehicle) +
                           $available(rweight-load-v(vehicle)) <=
                           weight-cap-r(route) ] ] )

// ----------------------------- Loading packages -----------------------------

#control :name "only-load-packages-into-reasonable-vehicles"
        forall t, package, vehicle [
               [t] !at-packagev(package, vehicle) ->
               ([t+1] !at-packagev(package, vehicle)) |
               exists location [
                      [t] at-packagel-generalized(package, location) &
                      [t] reasonable-vehicle-for-package(package,
                                                         vehicle,
                                                         location) ] ]

#define [t] reasonable-vehicle-for-package(package, vehicle, location-from):
        [t] typevp(vehicle, truck) &
        [t] reasonable-truck-for-package(package, vehicle, location-from) |
        [t] !typevp(vehicle, truck) &
        [t] reasonable-nontruck-for-package(package, vehicle, location-from)

// A truck is a reasonable mean of transportation for a package if:
#define [t] reasonable-truck-for-package(package, vehicle, location-from):
       // The package needs to go somewhere and the truck is empty.
       exists location-goal [
              goal(delivered(package, location-goal)) &
              ([t] location-from != location-goal) &
              !exists package2 [
                     package2 != package &
                     [t] at-packagev(package2, vehicle) ] &
              // The truck can deliver it or take it to a transportation center
              // from a non transportation center.
              (([t] can-go-by-truck(vehicle, location-from, location-goal) &
                [t] h-start(package) ->
                    tcenter(location-from) & !tcenter(location-goal)) |
               [t] !tcenter(location-from) &
               ([t] !t-end(package)) &
               exists location3 [
                      location3 != location-from &
                      [t] in-same-city(location3, location-from) &
                      [t] tcenter(location3) &
                      [t] can-go-by-truck(vehicle, location-from, location3) ])]
```

```
// A train or plane is a reasonable mean of transportation for a package if:
#define [t] reasonable-nontruck-for-package(package, vehicle, location-from):
       // The package needs to go to a location in the same city which is
       // reachable.
       exists location-goal [
              goal(delivered(package, location-goal)) &
              location-from != location-goal &
              [t] in-same-city(location-from, location-goal) &
              [t] can-go-by-nontruck(vehicle, location-from, location-goal) ] |
       // The package needs to go to another city that is reachable.
       exists location-goal, location3 [
              goal(delivered(package, location-goal)) &
              location-from != location-goal &
              [t] !in-same-city(location-from, location-goal) &
              [t] in-same-city(location3, location-goal) &
              [t] can-go-by-nontruck(vehicle, location-from, location3) ] |
       // The package needs to go to a third city that is a transportation hub.
       exists location-goal, location3 [
              goal(delivered(package, location-goal)) &
              location-from != location-goal &
              [t] !in-same-city(location-from, location-goal) &
              [t] !in-same-city(location3, location-goal) &
              [t] !hub(location-from) &
              [t] hub(location3) &
              ([t] can-go-by-nontruck(vehicle, location-from, location3)) &
              !exists location4 [
                     [t] in-same-city(location4, location-goal) &
                     [t] can-go-by-nontruck(vehicle,
                                            location-from,
                                            location4) ] ]

#control :name "only-unload-packages-after-moving-and-at-reasonable-locations"
        forall t, package, vehicle, location [
               [t] at-vehicle(vehicle, location) &
               [t] at-packagev(package, vehicle) &
               ([t+1] !at-packagev(package, vehicle)) ->
               [t] move(package) &
               (goal(delivered(package, location)) |
                [t] tcenter(location)) ]

#control :name "only-put-down-packages-if-they've-moved"
        forall t, package, location [
               [t] !at-packagel(package, location) &
               ([t+1] at-packagel(package, location)) ->
               [t] move(package) ]

// If a crane picks up a package from a vehicle, it must then put it on the
// ground instead of loading it into the vehicle again.
#control :name "put-packages-down-after-picking-them-up-from-vehicle"
         forall t, package, crane [
                [t] move(package) &
                [t] at-packagec(package, crane) &
                [t+1] !at-packagec(package, crane) ->
                [t+1] !move(package) ]

// Only connect chute if needed to load or unload a package.
#control :name "only-connect-chute-if-needed"
         forall t, vehicle [
                [t] !chute-connected(vehicle) &
                ([t+1] chute-connected(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
```

```
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                       exists package [
                              [t] at-packagev(package, vehicle) &
                              [t] need-to-unload-package-at(package,
                                                            location) ]) ] ]

// Only disconnect chute if there are no packages to load.
#control :name "only-disconnect-chute-if-not-needed"
         forall t, vehicle [
                [t] chute-connected(vehicle) &
                ([t+1] !chute-connected(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       !exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                                  need-to-move-package-from(package,
                                                           location) ] ] ]

// Only attach conveyor ramp if needed to load or unload a package.
#control :name "only-attach-conveyor-ramp-if-needed"
         forall t, vehicle, plane-ramp [
                [t] !ramp-connected(vehicle, plane-ramp) &
                ([t+1] ramp-connected(vehicle, plane-ramp)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              [t] reasonable-nontruck-for-package(package,
                                                                  vehicle,
                                                                  location) ] |
                        exists package [
                              [t] at-packagev(package, vehicle) &
                              [t] need-to-unload-package-at(package,
                                                            location) ]) ] ]

// Only lower conveyor ramp if needed to load or unload a package.
#control :name "only-lower-ramp-if-needed"
         forall t, vehicle [
                [t] !ramp-down(vehicle) &
                ([t+1] ramp-down(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                        exists package [
                              [t] at-packagev(package, vehicle) &
                              [t] need-to-unload-package-at(package,
                                                            location) ]) ] ]

// Only raise conveyor ramp if there are no packages to load or unload.
#control :name "only-raise-ramp-if-not-needed"
         forall t, vehicle [
                [t] ramp-down(vehicle) &
                ([t+1] !ramp-down(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       !(exists package [
                              [t] at-packagel-generalized(package, location) &
```

```
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                         exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// Only open vehicle door if needed to load or unload a package.
#control :name "only-open-door-if-needed"
         forall t, vehicle [
                [t] !door-open(vehicle) &
                ([t+1] door-open(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              [t] reasonable-vehicle-for-package(package,
                                                                 vehicle,
                                                                 location) ] |
                        exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// Only close vehicle door if there are no packages to load or unload.
#control :name "only-close-door-if-not-needed"
         forall t, vehicle [
                [t] door-open(vehicle) &
                ([t+1] !door-open(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       !(exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              [t] reasonable-vehicle-for-package(package,
                                                                 vehicle,
                                                                 location) ] |
                         exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// Only connect hose if needed to load or unload a package.
#control :name "only-connect-hose-if-needed"
         forall t, vehicle [
                [t] !hose-connected(vehicle) &
                ([t+1] hose-connected(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                        exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// Only disconnect hose if there are no packages to load.
#control :name "only-disconnect-hose-if-not-needed"
         forall t, vehicle [
                [t] hose-connected(vehicle) &
                ([t+1] !hose-connected(vehicle)) ->
```

```
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       !exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] ] ]

// Only open tanker valve if needed to load or unload a package.
#control :name "only-open-valve-if-needed"
         forall t, vehicle [
                [t] !valve-open(vehicle) &
                ([t+1] valve-open(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       (exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                        exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// Only close tanker valve if there are no packages to load or unload.
#control :name "only-close-valve-if-not-needed"
         forall t, vehicle [
                [t] valve-open(vehicle) &
                ([t+1] !valve-open(vehicle)) ->
                exists location [
                       ([t] at-vehicle(vehicle, location)) &
                       !(exists package [
                              [t] at-packagel-generalized(package, location) &
                              [t] package-vehicle-compatible(package, vehicle) &
                              need-to-move-package-from(package, location) ] |
                         exists package [
                               [t] at-packagev(package, vehicle) &
                               [t] need-to-unload-package-at(package,
                                                             location) ]) ] ]

// -------------------------------- General --------------------------------

// The package is at the location or being lifted by a crane at the location.
#define [t] at-packagel-generalized(package, location):
        ([t] at-packagel(package, location)) |
        exists crane [
               [t] at-equipment(crane, location) &
               [t] at-packagec(package, crane) ]

// The package has a goal to be at another location.
#define [t] in-wrong-city(package, location):
        exists location-to [
               goal(delivered(package, location-to)) &
               !exists city [
                      [t] in-city(location, city) &
                      [t] in-city(location-to, city) ] ]

// Location1 and location2 are in the same city.
#define [t] in-same-city(location1, location2):
        exists city [
               [t] in-city(location1, city) &
               [t] in-city(location2, city) ]

// The package is compatible with the vechile.
#define [t] package-vehicle-compatible(package, vehicle):
```

```
        exists ptype, vtype [
               [t] typep(package, ptype) &
               [t] typev(vehicle, vtype) &
               [t] pv-compatible(ptype, vtype) ]

// A goal forces the package to be moved.
#define [t] need-to-move-package-from(package, location):
        exists location2 [
               location != location2 &
               goal(delivered(package, location2)) ]

// The package has reached its destination or must be unloaded before further
// transportation.
#define [t] need-to-unload-package-at(package, location):
        goal(delivered(package, location)) |
        [t] move(package)

// Packages that are over(package) cannot be moved and should not be loaded
// into a vehicle.
#control :name "Don't-load-packages-that-are-over"
         forall t, package, vehicle [
                [t] !at-packagev(package, vehicle) ->
                ([t+1] !at-packagev(package, vehicle)) |
                [t] !over(package) ]

#control :name "only-deliver-if-goal"
         forall t, package, location [
                [t] !delivered(package, location) &
                [t+1] delivered(package, location) ->
                goal(delivered(package, location)) ]
```