Kongming: A Generative Planner for Hybrid Systems with Temporally Extended Goals

by

Hui X. Li

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Hui X. Li
April 9, 2010

Certified by .....
Prof. Brian Williams
Thesis Supervisor

Certified by .....
Dr. Andreas Hofmann
Thesis Committee Member

Certified by .....
Prof. Patrick Winston
Thesis Committee Member

Certified by .....
Prof. Rodney Brooks
Thesis Committee Member

Accepted by .....
Prof. Eytan Modiano
Chairman, Department Committee on Graduate Students

Kongming: A Generative Planner for Hybrid Systems with Temporally Extended Goals

by

Hui X. Li

Thesis supervisor: Brian Williams

Abstract

Most unmanned missions in space and undersea are commanded by a "script" that specifies a sequence of discrete commands and continuous actions. Currently such scripts are mostly hand-generated by human operators. This introduces inefficiency, puts a significant cognitive burden on the engineers, and prevents re-planning in response to environment disturbances or plan execution failure. For discrete systems, the field of autonomy has elevated the level of commanding by developing goal-directed systems, to which human operators specify a series of temporally extended goals to be accomplished, and the goal-directed systems automatically output the correct, executable command sequences. Increasingly, the control of autonomous systems involves performing actions with a mix of discrete and continuous effects. For example, a typical autonomous underwater vehicle (AUV) mission involves discrete actions, like get GPS and take sample, and continuous actions, like descend and ascend, which are influenced by the dynamical model of the vehicle. A hybrid planner generates a sequence of discrete and continuous actions that achieve the mission goals.

In this thesis, I present a novel approach to solve the generative planning problem for temporally extended goals for hybrid systems, involving both continuous and discrete actions. The planner, Kongming[1], incorporates two innovations. First, it employs a compact representation of all hybrid plans, called a Hybrid Flow Graph, which combines the strengths of a Planning Graph for discrete actions and Flow Tubes for continuous actions. Second, it engages novel reformulation schemes to handle temporally flexible actions and temporally extended goals. I have successfully demonstrated controlling an AUV in the Atlantic ocean using mission scripts solely generated by Kongming. I have also empirically evaluated Kongming on various real-world scenarios in the underwater domain and the air vehicle domain, and found it successfully and efficiently generates valid and optimal plans.

[1]Kongming is the courtesy name of the genius strategist of the Three Kingdoms period in ancient China.

献给

我的姥姥和父母

Acknowledgments

I would like to express my gratitude to all who have supported me in completion of this thesis.

First, I am grateful to my advisor, Brian Williams, for his guidance and encouragement on my research, for his working so hard with me to make the thesis deadline, and for his help and support when I could not enter the US for over a year.

I would like to thank my committee members and readers, Andreas Hofmann, Hans Thomas, Jim Bellingham, Maria Fox, Patrick Winston, Paul Robertson, and Rodney Brooks, for giving me insightful feedback on my thesis and defense. Especially I want to thank Andreas and Hans for the valuable discussions we had on my thesis, Maria for spending time going into technical depth of my work and giving me detailed feedback, and Patrick for giving me great advice on my defense presentation.

I would like to thank Justin Eskesen from the AUV Lab at MIT Sea Grant for providing Odyssey IV and helping us demonstrate Kongming on it. I had a great time swimming with the cool robot. I would like to thank Scott Smith and Ron Provine from Boeing for their help and support over these years.

I would like to thank my labmates in MERS for their help and support. Especially I want to thank Hiro, Patrick, Henri, David, Shannon and Alborz for the insightful discussions we had on my research. I would also like to thank everyone who came to my defense practices. The feedback was very helpful.

I would not have made it without the support and love from my parents and friends. They helped me go through the extremely tough moments in my life, and helped me grow stronger. I am truly grateful!

My research was funded by the Boeing Company under contract MIT-BA-GTA-1.

Contents

Chapter 1

Introduction

Contents

1.1 Motivation

Most unmanned missions in space and undersea are commanded by a "script" that specifies a sequence of discrete commands and continuous actions. Currently such

scripts are mostly hand-generated by human operators. For example, a large number of undersea missions for ocean observing are conducted by oceanographic research institutes, such as Monterey Bay Aquarium Research Institute (MBARI) [Ins], MIT Sea Grant [auv] and Woods Hole Oceanographic Institution (WHOI) [who]. At MBARI, marine scientists use autonomous underwater vehicles (AUVs) as one means to survey regions of the ocean, or map the seafloor, for example, in the Monterey Canyon. The missions are commanded by mission scripts consisting of pre-defined action sequences, specified by engineers. From conversations with MBARI engineers, I learned that a typical survey mission is roughly 20 hours long and involves roughly 200 behaviors. The engineers must pinpoint on a map every intermediate way point for the AUV to surface, in order to get GPS information every 30 minutes, calculate the depth value for each way point using navigation software, and generate the complete action sequence.

This introduces inefficiency, the potential for human error, and puts a significant cognitive burden on the engineers. These problems can be reduced, if we elevate the level of commanding, such that the operators only need to specify a set of goals that they want the AUV to accomplish, and have a planner produce the series of actions that achieve the mission goals. This is *generative* planning. Such a planner can enable online re-planning, by generating the action sequence online according to changes in the mission goals, changes or disturbances in the environment, or plan execution failure.

In the purely discrete-state domain, where states are discrete and actions only have discrete effects, the field of autonomy has elevated the level of commanding by developing goal-directed planning systems [WN97, Chu08]. The mission objectives are described as a desired evolution of goal states to the goal-directed planning systems. Given a description of the available operators to the planner, it automatically generates a correct, executable command sequence. The desired evolution of goal states is called a *temporally extended goal.*

Although a purely discrete-state abstraction is sometimes adequate to model and control an autonomous system, for example, the engine subsystem of a spacecraft

[WN97], almost no embedded system is purely discrete in reality. Often a purely discrete model does not suffice to control an autonomous system. For example, we need to be able to reason about the continuous effect of an unmanned air vehicle's fly action, or the continuous effect of a robotic arm turning its joint. Most embedded systems, are hybrid systems. They have a mix of discrete and continuous states, and can perform a mix of discrete and continuous actions. For example, an AUV can have `sampleTaken=true` as its discrete state and the $\langle x, y, z \rangle$ position as its continuous state; it can perform discrete actions like `take sample` and continuous actions like `descend`.

Classical AI planning has been well studied for the past few decades. [BF97, KS99, KS92, DK01a, HN01, BG99, LF99, KNHD97, PW92] are all generative planners. Later on a number of planners have been developed to handle time and other resources [SW99, LF02, GS02, DK01b, HG01, CFLS08b, Hof03]. These planners provide a substantial foundation for the field of AI planning, but cannot reason about actions that cause continuous changes in state space. There has been work in planning for actions with continuous changes [GSS04, WW99, PW94, McD03, CCFL09, CFLS08a, SD05], however, because they assume a fixed rate of change and a decoupled state transition, none of them can handle systems with complex dynamics, such as a second-order acceleration bounded system.

The planning problem I am interested in is how to develop a goal-directed, domain-independent planner for hybrid systems, such that human operators describe to the planner the temporally extended goals that they intend for a hybrid system to accomplish, and the planner elaborates a sequence of discrete commands and continuous actions for the hybrid system to perform to achieve the goals, based on the model of the hybrid system. The model of the system generally includes the discrete and continuous action types the system can perform, as well as the continuous dynamic model.

1.2 Problem Statement

The planning problem I am interested in is: given

- *initial conditions*,
- *temporally extended goals*,
- *hybrid durative action types*,
- *external constraints*, and
- an *objective function*,

find an optimal sequence of actions and state trajectory from the given action types that achieve the temporally extended goal without violating any of the constraints.

Next I explain these terms in the following subsections.

1.2.1 Input

In this subsection, I describe an example in order to intuitively demonstrate the different components of the input. The precise definitions are described in Chapter 3. The example is based on an MBARI (Monterey Bay Aquarium Research Institute) science mission scenario.

The scientists define the mission goals as follows:

> *As seen in the map in Fig. 1-1, first take samples in the A and B regions at depth 20 meters, and then in the C, D and E regions at depth 16 meters.*

The operation engineers add temporal bounds on the mission goals, as shown in Fig. 1-2. Engineers also add the following based on the environment and ship/fishing activities of the day:

> *Avoid traffic and fishing activities, characterized by the shaded polygon in Fig. 1-1. Maximize the time on surface to have maximum GPS coverage.*

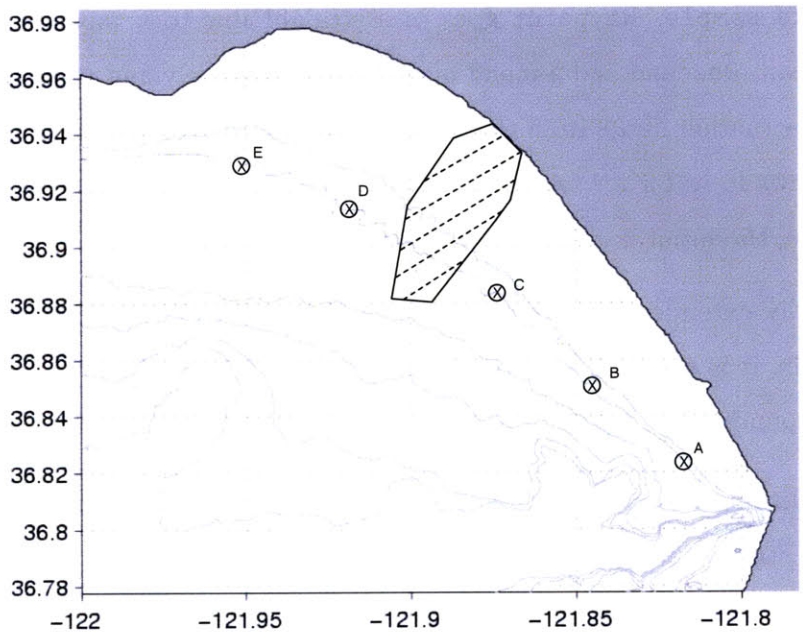


Figure 1-1: Map for the sampling mission in Monterey Bay. Grey area is land, and white area is ocean. Engineers characterize the area with traffic and fishing activities with a shaded polygon in the map. It is the obstacle that the AUV needs to avoid. *Picture courtesy of MBARI.*

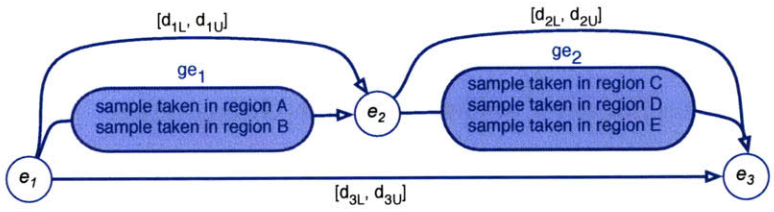


Figure 1-2: A QSP example for the underwater scenario associated with Fig. 1-1. Small circles represent events or time points. Purple shapes represent goals. Square brackets specify lower and upper temporal bounds between events.

The initial conditions are that the AUV starts from location (-121.8, 36.8), and no sample has yet been taken. There are four types of actions: `waypoint, descend, ascend`, and `take sample`. `Waypoint` goes in a straight line to a waypoint on the surface of the ocean. `Descend` and `ascend` move both horizontally and vertically; they involve different actuation limits from the `waypoint` action as well as from each other. `Take sample` draws in water for sampling.

To summarize, the input is described as follows:

- A temporally extended goal, expressed as a Qualitative State Plan (QSP). In the example, it is the evolution of goal states specified by the scientists, with temporal bounds augmented by engineers, shown pictorially in Fig. 1-2.

- Initial conditions. In the example, they are the start location of the AUV and its discrete state at the start of the mission.

- Model of each action type. In the example, they are the duration bounds, conditions, effects and dynamic model of `waypoint, descend, ascend`, and `take sample`. The actions can have both continuous and discrete effects. `Waypoint, descend` and `ascend` actions in the example all have continuous effects. Their dynamics are defined by different state transition equations and different bounds on control inputs. `Take sample` has a discrete effect - sample taken.

- External constraints. In the example, they are staying outside the shaded polygon in Fig. 1-1 for avoiding traffic and fishing activities.

- An objective function. In the example, it is to maximize the time that the AUV spends on the surface, in order to have maximum GPS coverage. Alternatively it could be to minimize distance traveled, minimize mission time, minimize battery usage, or other user-defined linear/quadratic objectives.

1.2.2 Output

In this section, I describe the different components of the planner output using our example. The precise definitions are described in Chapter 3.

The output is a sequence of actions formed from the four action types described in the input example, along with an optimal trajectory. It is visualized in Fig. 1-3. The AUV first goes to the center of region A, then descends to 20 meters, takes a sample, and ascends to the surface. The AUV then goes to the center of region B, and repeat the actions until all goals are achieved.

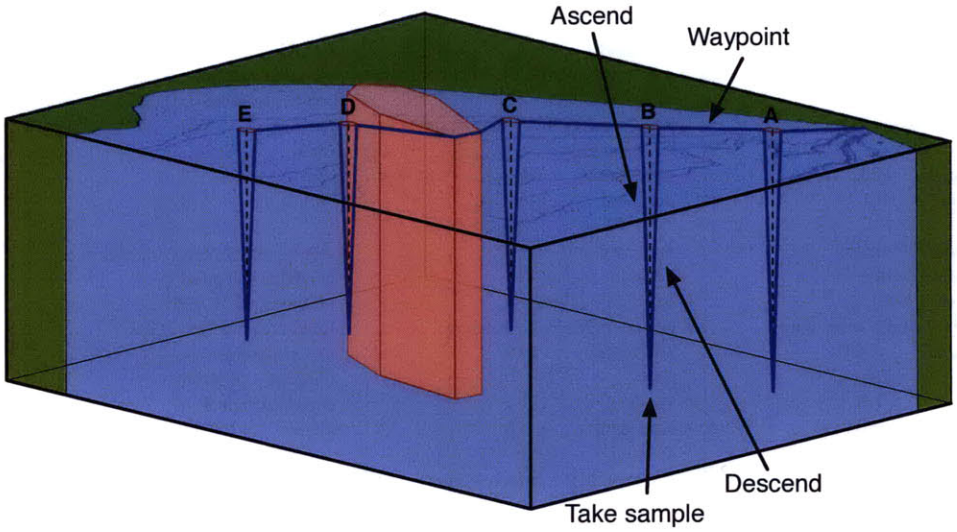


Figure 1-3: An output example for the underwater scenario corresponding to Fig. 1-1.

The output can also be described as a mission script, as shown in Fig. 1-4.

The mission scripts for MBARI science missions normally last roughly 20 hours and involve roughly 200 behaviors. Hence, the scripts are significantly longer than the example mission script shown here. However, the number of action **types** in the MBARI scripts is similar to the number of action types in the example, which is 4.

More generally, the output of the planner is an optimal partial order hybrid plan. The hybrid plan consists of an action sequence and a state trajectory that satisfy the temporally extended goal, the initial condition, and the constraints. The plan is also optimal according to the objective function given the number of time steps in the plan.

```
# Waypoint 1 (center of region A)
behavior waypoint {
latitude    = 36.8225;
longitude   = -121.8210;
captureRadius = 5;
duration    = 2000;
speed       = 1.5;
depth  = 0.0;}

# Descend to 20 meters
behavior descend {
horizontalMode  = heading;
horizontal      = 321;
pitch       = -15;
speed       = 1.5;
maxDepth    = 20;
duration    = 240;}

# Gulp behavior to take samples
behavior gulp {
maxDepth    = 20;
duration    = 10;}

# Ascend to 0 meters
behavior ascend {
duration = 1200;
horizontalMode  = heading;
horizontal      = 321;
pitch           = 20;
speed           = 1.5;
endDepth        = 0.0;}

# Waypoint 2 (center of region B)
behavior waypoint {
latitude    = 36.8509;
longitude   = -121.8485;
captureRadius = 5;
duration    = 1800;
speed       = 1.5;
depth       = 0.0;}

# Descend to 20 meters
behavior descend {
horizontalMode  = heading;
horizontal      = 328;
pitch       = -15;
speed       = 1.5;
maxDepth    = 20;
duration    = 240;}

# Gulp behavior to take samples
behavior gulp {
maxDepth    = 20;
duration    = 10;}

# Ascend to 0 meters
behavior ascend {
duration = 1200;
horizontalMode  = heading;
horizontal      = 331;
pitch           = 20;
speed           = 1.5;
endDepth        = 0.0;}
```

```
# Waypoint 3 (center of region C)
behavior waypoint {
latitude    = 36.8827;
longitude   = -121.8751;
captureRadius = 5;
duration    = 2100;
speed       = 1.5;
depth  = 0.0;}

# Descend to 16 meters
behavior descend {
horizontalMode  = heading;
horizontal      = 313;
pitch       = -15;
speed       = 1.5;
maxDepth    = 16;
duration    = 180;}

# Gulp behavior to take samples
behavior gulp {
maxDepth    = 16;
duration    = 10;}

# Ascend to 0 meters
behavior ascend {
duration = 1200;
horizontalMode  = heading;
horizontal      = 315;
pitch           = 20;
speed           = 1.5;
endDepth        = 0.0;}

# Waypoint 4 (obstacle corner)
behavior waypoint {
latitude    = 36.8801;
longitude   = -121.8907;
captureRadius = 5;
duration    = 1600;
speed       = 1.5;
depth       = 0.0;}

# Waypoint 5 (obstacle corner)
behavior waypoint {
latitude    = 36.8801;
longitude   = -121.9112;
captureRadius = 5;
duration    = 1600;
speed       = 1.5;
depth       = 0.0;}

# Waypoint 6 (center of region D)
behavior waypoint {
latitude    = 36.9150;
longitude   = -121.9251;
captureRadius = 5;
duration    = 2100;
speed       = 1.5;
depth       = 0.0;}
```

```
# Descend to 16 meters
behavior descend {
horizontalMode  = heading;
horizontal      = 347;
pitch       = -15;
speed       = 1.5;
maxDepth    = 16;
duration    = 180;}

# Gulp behavior to take samples
behavior gulp {
maxDepth    = 16;
duration    = 10;}

# Ascend to 0 meters
behavior ascend {
duration = 1200;
horizontalMode  = heading;
horizontal      = 331;
pitch           = 20;
speed           = 1.5;
endDepth        = 0.0;}

# Waypoint 7 (center of region E)
behavior waypoint {
latitude    = 36.9295;
longitude   = -121.9508;
captureRadius = 5;
duration    = 2000;
speed       = 1.5;
depth       = 0.0;}

# Descend to 16 meters
behavior descend {
horizontalMode  = heading;
horizontal      = 323;
pitch       = -15;
speed       = 1.5;
maxDepth    = 16;
duration    = 180;}

# Gulp behavior to take samples
behavior gulp {
maxDepth    = 16;
duration    = 10;}

# Ascend to 0 meters
behavior ascend {
duration = 1200;
horizontalMode  = heading;
horizontal      = 328;
pitch           = 20;
speed           = 1.5;
endDepth        = 0.0;}
```

Figure 1-4: Mission script for the example mission, without the initialization and safety behaviors.

1.3 Challenges

There are two key challenges to solving the problem stated in the previous section. I summarize them here and elaborate in the next two sections. First, in order to plan with hybrid actions, which have both continuous and discrete effects, we need a representation for the infinite number of state trajectories associated with the continuous action effects. In addition, which hybrid actions to take at any time is a combinatorial choice. It is a challenge to find a representation for this complex plan space, where the infinite number of trajectories and the combinatorial choices are combined. Second, in order to plan for temporally extended goals instead of simply final goals, we need to ensure that the temporal constraints associated with the goals are satisfied by the output plan. In addition, the hybrid actions have flexible temporal bounds on their durations. The temporal constraints and flexibility pose a challenge to solving our planning problem. I next discuss each challenge in more detail.

1.3.1 Complex Plan Space

In traditional AI planning, there are only discrete actions and propositional states. Hence, the choices to make are discrete or finite domain, for example, in playing chess. On the other hand, in control and path planning, there are only continuous actions and real-valued states. For example, navigating a vehicle [Léa05], or controlling a biped [Hof06]. However, the planning problem I am solving involves both.

A straightforward way is to encode the hybrid planning problem directly into a mixed discrete/continuous optimization problem, and to solve it using a state-of-the-art constraint solver. Doing this, however, puts all the computational burden on the solver. Our experiments in Chapter 8 show that the search space is prohibitive for even small problems. Therefore, I need a compact representation of the complex plan space, where the infinite number of trajectories and the finite domain choices are combined. The representation should contain all the valid plans and prune a large number of invalid plans, so that the search space becomes manageable.

1.3.2 Temporal Constraints and Flexibility

Temporally extended goals not only require the goal states to be achieved, but also require them to be achieved at different times, over extended durations, in the right order, and within the right temporal bounds. Moreover, it is unrealistic to fix the durations of hybrid action types in the problem specification. For example, taking a sample in the ocean may take from 5 to 30 seconds; and gliding on the surface of the ocean may take arbitrarily long until the destination is reached or battery runs out.

These temporal constraints and flexibility add an extra complexity to our hybrid planning problem. When goals and actions have flexible time bounds, there potentially are an infinite number of choices for their duration value.

I next discuss our approach to addressing the above described challenges.

1.4 Approach and Innovations

My solution to the stated problem is a planning system, called Kongming. Kongming is a domain-independent generative planner that is capable of planning for temporally extended goals for a range of hybrid autonomous systems.

Kongming addresses the two challenges described in Section 1.3 with two key innovations. To address the first challenge (complex plan space), Kongming introduces a compact representation of the complex plan space, called a Hybrid Flow Graph. It provides a natural way of representing continuous trajectories in a discrete planning framework. It combines the strengths of a Planning Graph from Graphplan [BF97] for discrete actions and Flow Tubes [Hof06] for continuous actions. The Hybrid Flow Graph contains all the valid plans while pruning many invalid plans. To address the second challenge (temporal constraints and flexibility), Kongming introduces two reformulation schemes to encode temporally extended goals as actions and final goals, and to encode actions with flexible time bounds as actions with fixed and equal durations. These reformulation schemes pre-process the temporal reasoning part of our hybrid planning problem, and reduce complexity.

I now describe these two innovations in more detail.

1.4.1 Hybrid Flow Graph

When planning with purely discrete actions, as in Graphplan [BF97] and other discrete planners, the decision variables are discrete, and hence the planner only needs to reason about a finite number of trajectories in plan space. However, when continuous actions and continuous states are included in planning, the planner needs a compact way of representing and reasoning about an infinite number of trajectories and states. My representation of the plan space is called a Hybrid Flow Graph. It builds upon the Planning Graph [BF97] and flow tubes [Hof06].

Planning Graph

The Planning Graph introduced in Graphplan [BF97], is a compact structure for representing the plan space in the STRIPS-like planning domain. It has been employed by a wide range of modern planners, including LPGP [LF02], STAN [LF99], GP-CSP [DK01a], to name a few. The Graphplan planner constructs, analyses the graph structure, and extracts a valid plan. A Planning Graph encodes the planning problem in such a way that many useful constraints inherent in the problem become explicitly available to reduce the amount of search needed. Planning Graphs can be constructed quickly: they have polynomial size and can be built in polynomial time.

A Planning Graph is a directed, leveled graph, which alternates between proposition levels, containing propositions, and action levels, containing action instances. Proposition level i contains all propositions that can be true at time i, and action level i contains all possible actions to take place at time i. Fig. 1-5 shows an example of a Planning Graph.

Graphplan identifies mutual exclusion relations among propositions and actions. The mutual exclusions make constraints inherent in the planning problem explicit. Two actions at a given action level are *mutually exclusive* if no valid plan can contain both. Similarly, two propositions at a given proposition level are *mutually exclusive* if no valid plan can make both true. Identifying the mutual exclusions helps reduce the search for a subgraph of the Planning Graph that corresponds to a valid plan. Intuitively, this is

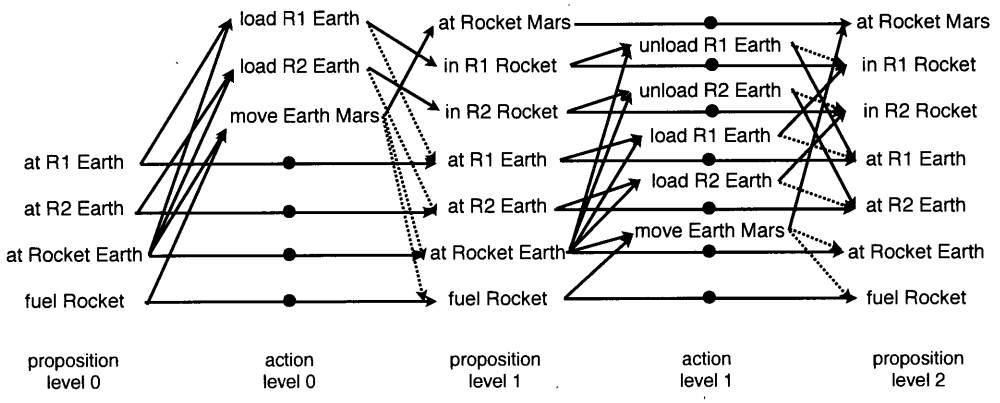


Figure 1-5: A Planning Graph of 3 proposition levels and 2 action levels. Round black dots represent no-op (do nothing) actions. Solid lines connect conditions with actions and actions with add-effects. Dotted lines connect actions with delete-effects.

because an action is not added to an action level if two of its conditions are identified as mutually exclusive in the previous proposition level. The mutual exclusion relations are discovered and propagated through the Planning Graph using a few rules. They prove to be greatly helpful in pruning search space [BF97, KS99].

The Planning Graph works well for discrete planning, but it cannot represent actions that cause continuous changes in state space, as each action contains an infinite number of continuous state trajectories. Hence, I use flow tubes as an abstraction for the continuous state trajectories, and augment the flow tubes in the Planning Graph.

Flow Tube

Flow tubes have been used in the qualitative reasoning community to represent a set of trajectories with common characteristics that connect two regions. Zhao [Zha92][1] uses them to characterize phase space during analysis of nonlinear dynamical systems. Hofmann [Hof06] uses flow tubes to represent bundles of state trajectories that take into account dynamic limitations due to under-actuation, while satisfying plan requirements for the foot placement of walking bipeds. By defining the valid operating regions for the state variables and control parameters in the abstracted model of a biped, the flow tubes prune infeasible trajectories and ensure plan execution. For example, Fig. 1-6

[1]They are called flow pipes in [Zha92].

shows flow tubes for the center of mass of a biped. Similar representations are used in the robotics community, under the name funnels [Ted09], and in the hybrid systems community, under reachability analysis [KV07, KGBM04, Gir05, SK03, Kos01].

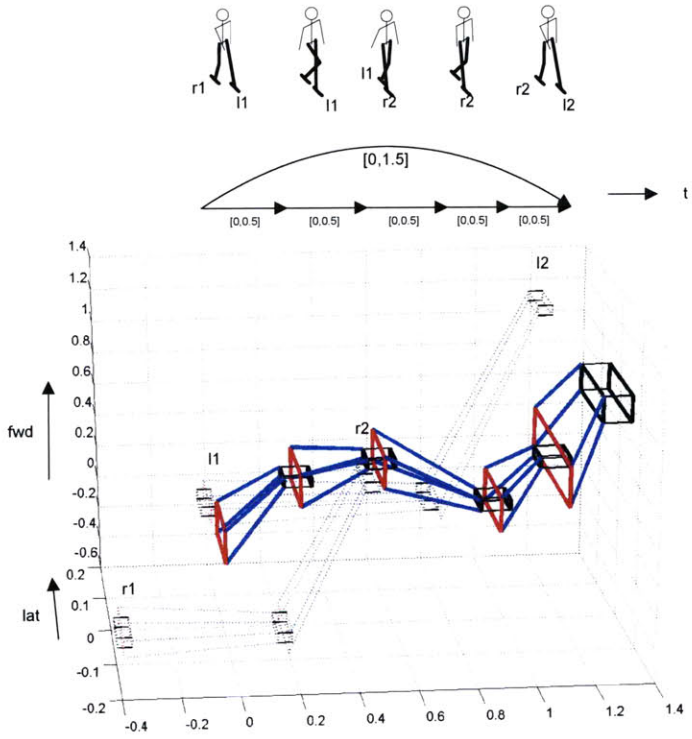


Figure 1-6: Flow tubes for center of mass of a biped are shown, with initial regions in red, goal regions in black, and tubes in blue. The flow tubes define permissible operating regions in state space. [Hof06]

The problem that Kongming uses a flow tube to solve is, given an initial region in state space, defined by a conjunction of linear (in)equalities over the state variables, given the model of a hybrid action, including a state equation and actuation limits of the system dynamics, and given a duration, generate an abstract description of all possible state trajectories of this action starting from the initial region. Fig. 1-7 shows an example of a flow tube, starting from an initial region R_I, and ending in an end region. The end region represents all the reachable states by performing the action for duration d from the initial region. Any cross-section cut through the flow tube at a specific time point represents all the reachable states by performing the action for the specific duration from the initial region.

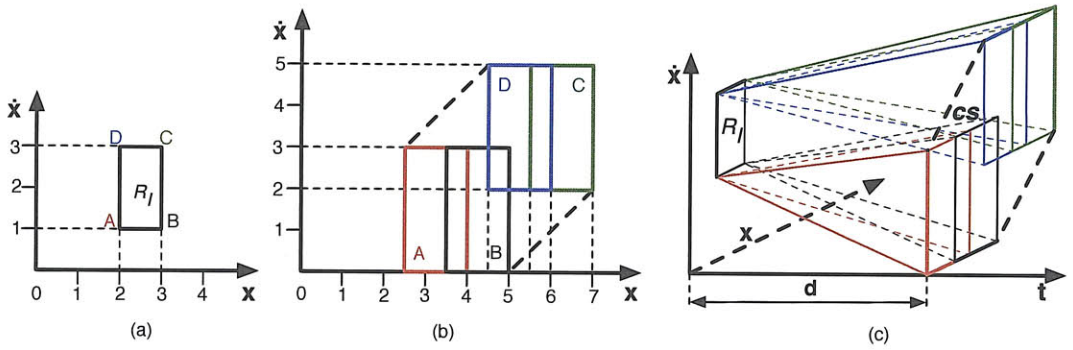


Figure 1-7: An example of a flow tube of a hybrid action in 1D for a second-order acceleration limited dynamical system. (a) shows the initial region R_I. (b) shows the end region. (c) shows the flow tube of duration d.

The flow tubes in Kongming are similar to those in [Hof06] in that Kongming also uses flow tubes to represent bundles of state trajectories that satisfy dynamic limitations and plan requirements. Flow tubes in Kongming are different in the following aspects. First, they contain all valid trajectories; whereas flow tubes in [Hof06] exclude some of the valid trajectories. Second, they also contain invalid trajectories; whereas flow tubes in [Hof06] only contain valid trajectories. Third, two flow tubes in Kongming are connected, if the end region of the first flow tube intersects with the condition of the second flow tube, whereas two flow tubes in [Hof06] are connected, if the end region of the first flow tube is a subset of the initial region of the second flow tube. The reason for the first three differences is because Kongming needs to have a complete representation of all valid plans in the Hybrid Flow Graph, so that it can search for an optimal plan using a constraint solver, whereas [Hof06] searches for valid rather than optimal trajectories. Finally, in Kongming, a flow tube can also be connected to a discrete action, whereas there are no discrete actions in [Hof06].

After introducing the two components of our representation of the complex plan space, a Hybrid Flow Graph, I next describe how they fit together.

Hybrid Flow Graph Expansion

The initial conditions of the hybrid planning problem form the first fact level of the Hybrid Flow Graph. The first fact level is followed by the first action level, which

contains all the actions whose conditions are satisfied in the first fact level. Then the effects of these actions form the next fact level. The Hybrid Flow Graph keeps expanding until the goal conditions are achieved.

Each hybrid action is represented by a flow tube, starting from an initial region and ending in an end region. In a Hybrid Flow Graph, one flow tube is connected to another, if the end region of the previous has a nonempty intersection with the continuous condition of the following. I choose having a nonempty intersection as the connecting criteria in order to keep the completeness of our plan representation. As shown in Fig. 1-8(a), the nonempty intersection is the initial region of the following flow tube. Conversely, as shown in Fig. 1-8(b), if the intersection is empty, then the following flow tube is not added into the Hybrid Flow Graph.

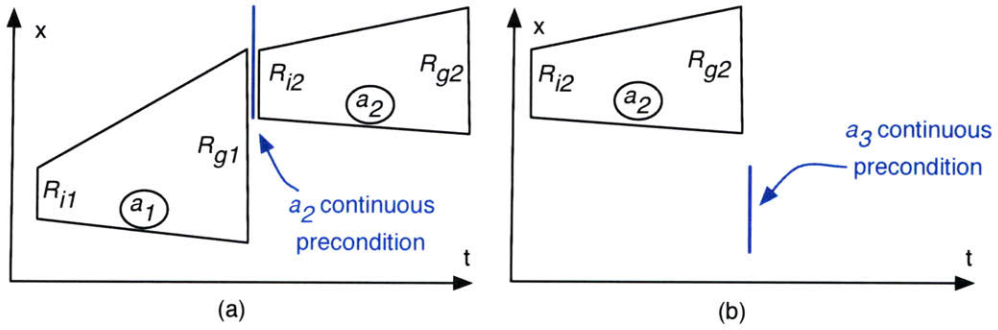


Figure 1-8: (a) Flow tube a_2 is connected to flow tube a_1 because the end region of a_1 has a nonempty intersection with the continuous condition of a_2. (b) Conversely, flow tube a_3 is not connected to flow tube a_2 because the end region of a_2 has an empty intersection with the continuous condition of a_3.

An example of a Hybrid Flow Graph is shown in Fig. 1-9. The Hybrid Flow Graph starts with *fact level* 1, followed by *action level* 1, and then *fact level* 2, followed by action level 2, and so on. A Hybrid Flow Graph example. Each fact level contains continuous regions (in blue) and literals (in black). Each action level contains hybrid actions. The hybrid actions with specified dynamics are represented by flow tubes (in blue, while the dynamics of some hybrid actions are unspecified (in black). Big black dots represent no-op actions. Arrows connect conditions to hybrid actions and hybrid actions to effects.

Similar to Graphplan, identifying and propagating mutual exclusion relations is

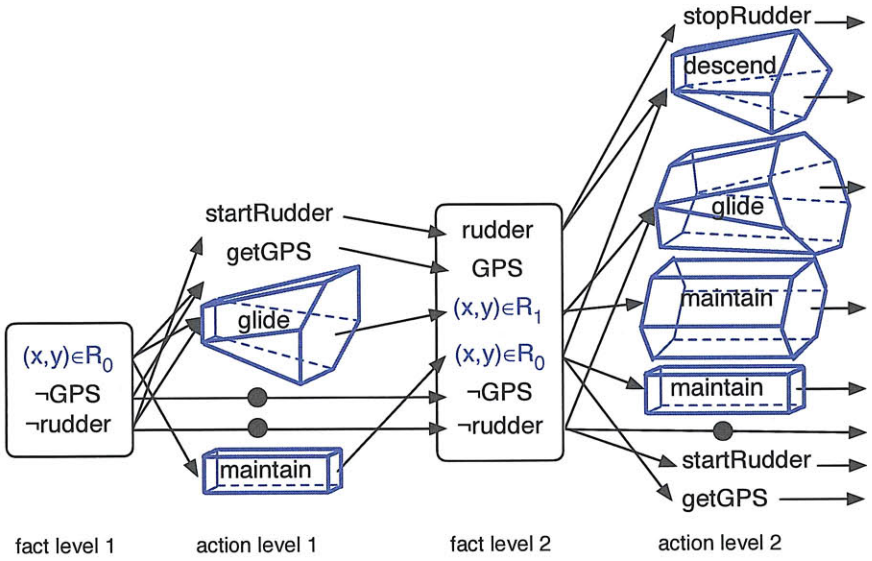


Figure 1-9: A Hybrid Flow Graph example. Each fact level contains continuous regions (in blue) and literals (in black). Each action level contains hybrid actions. The hybrid actions with specified dynamics are represented by flow tubes (in blue, while the dynamics of some hybrid actions are unspecified (in black). Big black dots represent no-op actions. Arrows connect conditions to hybrid actions and hybrid actions to effects.

an integral step in constructing and searching the Hybrid Flow Graph. The mutual exclusion relations can be largely useful in pruning invalid actions and invalid facts, and therefore invalid plans, in a Hybrid Flow Graph. They are propagated from one level to the next, using a set of rules. The rules in Kongming are generalized from those in Graphplan, because in a Hybrid Flow Graph fact levels not only contain propositional facts but also continuous regions, and action levels contain hybrid actions instead of purely discrete actions.

Kongming employs a constraint-based planner on the Hybrid Flow Graph to find an optimal plan. It encodes the Hybrid Flow Graph as mixed logical quadratic programs (MLQPs), and solves the MLQPs using a state-of-the-art MLQP solver, currently CPLEX.

The Hybrid Flow Graph is discussed in detail in Chapter 4, and the constraint-based planner is discussed in Chapter 5. They both follow the assumption that there is a final goal state, and that the hybrid actions have fixed and equal duration, which

I call hybrid *atomic* actions. However, the problem that Kongming solves contains hybrid actions with flexible time bounds, which I call hybrid *durative* actions, and a qualitative state plan (QSP) as goals, and hence, another important component in Kongming is the reformulations.

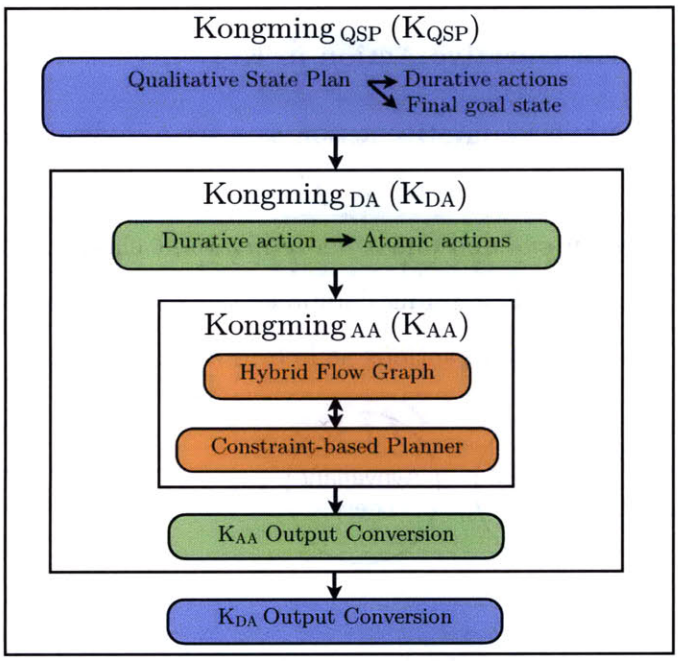


Figure 1-10: Overview of Kongming's approach.

As shown in the approach overview diagram in Fig. 1-10, the core of Kongming is a planner for atomic actions, which I call Kongming$_{AA}$ or K_{AA}. Kongming$_{DA}$ or K_{DA} is a planner that handles durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. Kongming$_{QSP}$ or K_{QSP} is a planner that plans for QSPs instead of a final goal state, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. The output of K_{AA}, K_{DA} and K_{QSP} are slightly different, as defined in Chapter 3. Hence, K_{DA} has the K_{AA} output conversion module used to process the output from K_{AA}, and similarly, K_{QSP} has the K_{DA} output conversion module used to process the output from K_{DA}.

1.4.2 Reformulations

I introduce the two layers of reformulation in this section. I first introduce the reformulation of hybrid durative actions to hybrid atomic actions in K_{DA}, and then introduce the reformulation of a QSP to hybrid durative actions and a final goal state in K_{QSP}.

Reformulating A Hybrid Durative Action in K_{DA}

Kongming reformulates a hybrid durative action as a set of hybrid atomic actions. This generalizes upon the encoding of LPGP [LF02]. LPGP encodes each discrete durative action as a *start*, an *end* and one or more invariant checking instantaneous actions, as shown in Fig. 1-11. Kongming combines the LPGP encoding with flow tubes to reformulate hybrid durative actions as hybrid atomic actions.

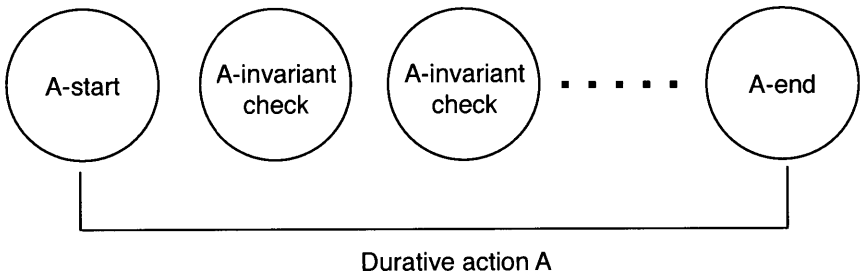


Figure 1-11: In LPGP [LF02], a durative action is formulated as a *start* action at the start, an *end* action at the end, and a series of actions for invariant checking in the middle.

Kongming generalizes the LPGP encoding to hybrid durative actions with flexible durations. Because the LPGP encoding only applies to actions with discrete conditions and effects, the main challenge is how to incorporate the flow tubes of the hybrid actions in the encoding.

Kongming encodes each hybrid durative action as a set of atomic actions, by "slicing" the flow tube into multiple pieces. All flow tube slices have the same duration, Δt. This is because Kongming allows flexible control input in the specification of hybrid action types. The constant length, Δt, for each flow tube slice ensures linearity in flow tube computation and the MLQP encoding.

Kongming combines the flow tube slices with the LPGP encoding, to ensure that first, the *start* flow tube slice is performed first, followed by one or more *intermediate* flow tube slices, and ending with the *end* flow tube slice; second, the conditions at various stages are enforced at the beginning of each flow tube slice; third, the effects at various stages are added at the end of each flow tube slice.

Reformulating A Qualitative State Plan in K_{QSP}

Kongming reformulates a QSP as hybrid durative actions and a final goal state. This is based on the idea that each goal state of a QSP can be achieved by specifying them in the conditions of an action. Kongming creates a hybrid durative action for each goal state of a QSP. The conditions of the action include all the state constraints on the goal state. In order to make the effects of the action true, the conditions have to be satisfied, which enforces the goal state to be achieved.

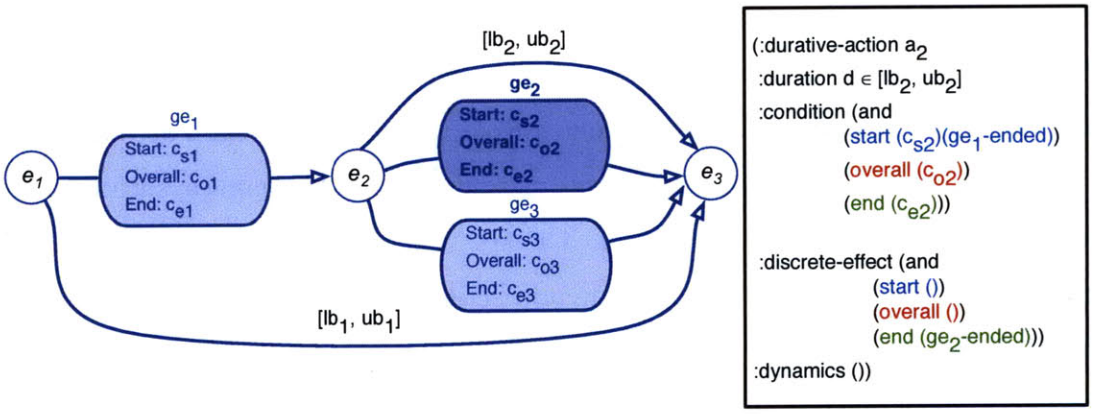


Figure 1-12: Hybrid durative action type a_2 is created for goal episode ge_2. ge_1-ended and ge_2-ended are literals, representing respectively the fact that ge_1 and ge_2 are achieved. c_{s2}, c_{i2} and c_{e2} are the constraints at different times of ge_2.

On the left-hand side of Fig. 1-12, there is a QSP, with three goal episodes. We focus on goal episode ge_2. The reformulation of ge_2 is shown on the right-hand side of Fig. 1-12. It shows the hybrid durative action created for goal ge_2. The duration of the action is bounded by the simple temporal constraint on the goal episode, if there is any. The conditions of the action include the state constraints of the goal episode, and the fact that all predecessors of the goal episode have been achieved. The discrete

effect of the action at the end is the fact that this goal episode has been achieved. Dynamics are unspecified.

The final goal state, which is another output of the reformulation, specifies that the end goal state in the QSP is achieved.

1.5 Contributions

There exist planners that are capable of planning for actions that cause continuous changes in state space [GSS04, WW99, PW94, McD03, CCFL09, CFLS08a, SD05, MBB+09], and a few of them [PW94, McD03, SD05, CCFL09, MBB+09] can handle actions with duration-dependent continuous effects. It is essential to be able to plan for actions with duration-dependent continuous effects, because such actions are pervasive in real-world planning problems. For example, the `fly` action of an air vehicle, the `descend` action of an underwater vehicle, or the `turn joint` action of an robotic arm. The changes in state space caused by such actions depend on the action duration.

However, all such existing planners assume a fixed rate of change and a decoupled state transition. None of them can handle systems with coupled dynamics, such as a second-order acceleration bounded system. The key benefit of our approach is that, Kongming allows complex dynamics of the hybrid actions through the introduction of flow tubes. Kongming is unique in that it is the first temporal planner that plans for autonomous systems, such as air, underwater or land vehicles, whose control input is not limited to constant and state transition is not limited to decoupled first order.

1.6 Thesis Outline

There are 9 chapters in total in this thesis. Chapter 2 is dedicated to related work. Chapter 3 formally states the planning problem and gives definitions of the input and output of Kongming. Chapter 4 through 7 cover the technical approach of Kongming. Chapter 4 introduces the compact representation of the complex plan space, called a Hybrid Flow Graph. Chapter 5 describes the constraint-based planning algorithm

that operates on the Hybrid Flow Graph representation. Chapter 6 introduces K_{DA} and its reformulation scheme for durative actions with flexible durations. Chapter 7 introduces K_{QSP} and its reformulation scheme for a QSP. Chapter 8 presents empirical results. Chapter 9 concludes and discusses future work.

Chapter 2

Related Work

Contents

2.1 Problem Representation Language

The Planning Domain Definition Language (PDDL) [MtAPCC98] was released in 1998, and has since become the standard language for planning competitions, such as the AI Planning Systems (AIPS) competitions and the International Planning Competitions. It is an action-centered language, inspired by the STRIPS [FN71] formulation of planning problems. PDDL2.1 [FL03] extends PDDL to include numeric expressions and durative actions. As shown in Fig. 2-1, in the action specification for pouring water between jugs, the precondition is that the empty volume in `jug2` is at least as much as the amount of water in `jug1`, and the effect is that the amount of

water in jug1 becomes 0 and the amount of water in jug2 increases by the original amount of water in jug1.

```
(define (domain jug-pouring)
   (:requirements :typing :fluents)
   (:types jug)
   (:functions
        (amount ?j - jug)
        (capacity ?j - jug))

   (:action pour
       :parameters (?jug1 ?jug2 - jug)
       :precondition (>= (- (capacity ?jug2) (amount ?jug2)) (amount ?jug1))
       :effect (and (assign (amount ?jug1) 0)
                    (increase (amount ?jug2) (amount ?jug1)))
)
```

Figure 2-1: An action example in PDDL2.1, showing the expression of numeric fluents [FL03]. "?jug1 ?jug2 - jug" means that jug1 and jug2 are of the type jug. "capacity ?jug2" means the capacity of jug2.

PDDL2.1 also extends PDDL with durative actions. Preconditions and effects are expressed at the start, in the middle, and at the end of an action. A continuous effect of a durative action is expressed as a linear function of time past while performing this action. As shown in Fig. 2-2, conditions and effects are specified for at start, over all and at end. The continuous effect of action fly is that the fuel-level decreases by the product of fuel-consumption-rate and t.

```
(:durative-action fly
     :parameters (?p - airplane ?a ?b - airport)
     :duration (= ?duration (flight-time ?a ?b))
     :condition (and (at start (at ?p ?a))
                     (over all (inflight ?p))
                     (over all (>= (fuel-level ?p) 0)))
     :effect (and (at start (not (at ?p ?a)))
                  (at start (inflight ?p))
                  (at end (not (inflight ?p)))
                  (at end (at ?p ?b))
                  (decrease (fuel-level ?p)
                            (* #t (fuel-consumption-rate ?p)))))
```

Figure 2-2: A durative action example in PDDL2.1 [FL03].

PDDL+ [FL01] provides an alternative to the continuous durative action model of PDDL2.1. The main difference is that PDDL+ separates changes to the world that

are directly enacted by the system under control from those indirect changes that are due to physical processes and their consequences. It also provides a formal semantics by mapping planning instances into constructs of hybrid automata.

The problem representation Kongming uses is PDDL-like. It is described in detail in Chapter 3. Kongming and PDDL2.1/PDDL+ share the same representation for the propositional part of initial conditions, action durations, action preconditions *at start*, *over all* and *at end*, and action effects *at start*, *over all* and *at end*. However, Kongming differs from PDDL2.1 and PDDL+ in that Kongming includes the following items:

- The **continuous precondition** of an action in Kongming's input is specified by a conjunction of linear (in)equalities over the state variables, $\wedge_i f_i(\mathbf{x}) \leq 0$. This is a polytope[1] in state space. PDDL2.1 or PDDL+ can only express the special case, where each $f_i(\mathbf{x})$ is a single-variable linear function, in other words, the polytope is an orthotope[2]. This generalization in Kongming is useful. For example, the action of putting out a forest fire requires the air vehicle to be over the fire region, which is not necessarily rectangular.

- The **state transition** of an action in Kongming's input is defined to be a linear nth-order state transition equation, $\forall t_i \ \mathbf{x}(t_i) = \mathbf{A}\mathbf{x}(t_{i-1}) + \mathbf{B}\mathbf{u}(t_{i-1})$, where $\mathbf{x}$ is state variables, $\mathbf{u}$ is control variables, and $\mathbf{A}$ and $\mathbf{B}$ are scalar matrices. In contrast, PDDL2.1 and PDDL+ are limited to 1st order decoupled state transition. For example, the second-order coupled state equation in Eq. 2.1 is expressible for Kongming, but not for PDDL2.1 or PDDL+. This generalization in Kongming is important. For example, the dynamics of an air vehicle normally cannot be modeled as linear first-order, but they can be modeled as linear

[1] A polytope is an n-dimensional region enclosed by a finite number of hyperplanes.
[2] An orthotope is an n-dimensional generalization of a rectangular region.

second-order [Léa05].

$$\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}(t_i) = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}(t_{i-1}) + \begin{bmatrix} \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^2}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \times \begin{bmatrix} a_x \\ a_y \end{bmatrix}(t_{i-1}) \tag{2.1}$$

- The **continuous initial condition** in Kongming's input is specified by a conjunction of linear (in)equalities over the state variables, $\wedge_i f_i(\mathbf{x}) \leq 0$, and hence, cannot be expressed in PDDL2.1 or PDDL+. The importance of this generalization is similar to the discussion earlier for the continuous preconditions.

- The **external constraints** in Kongming's input, for example, an obstacle avoidance requirement, is specified by a conjunction of disjunctions of linear inequalities over the state variables, $\wedge_i \vee_j f_{ij}(\mathbf{s}) \leq 0$. Such constraints cannot be expressed in PDDL2.1 or PDDL+. This generalization in Kongming is important. For example, it is unrealistic to assume all obstacles are rectangular.

- The **objective function** in Kongming's input is specified by a quadratic or linear function of the state variables. In PDDL2.1 or PDDL+, the objective function has to be linear. This generalization in Kongming is important. For example, in order to minimize the distance traveled, the objective function needs to be quadratic.

There are mainly two features in PDDL2.1 or PDDL+ that cannot naturally be expressed in Kongming's input: conditional effects, and increase/decrease continuous effects. A conditional effect is an effect of an action that only becomes true if specific conditions are satisfied. For example, the action `move suitcase` from the old location to a new location has the effect that, if the keys are in the suitcase, then they are in the new location. It has been proven that the Planning Graph [BF97] can be extended to handling conditional effects [ASW98], the approach of which can be employed by Kongming to extend the Hybrid Flow Graph.

The second feature, increase or decrease continuous effect, allows multiple actions to accumulatively alter the state of the system under control. For example, a bathtub can be filled and drained at the same time. The `fill` action has the effect of increasing the amount of water in the tub (the state of the system) at a specific rate r_1, and the `drain` action has the effect of decreasing the amount of water at a specific rate r_2. Accumulatively, the amount of water in the tub (the state of the system) increases at rate $r_1 - r_2$. This is different from Kongming. The continuous effects in Kongming's action description constrain the new state of the system, through the state transition equations in the action dynamics. Kongming allows multiple actions to take place simultaneously unless their effects constrain the state of the system to be in exclusive regions.

2.2 Planning with Continuous Change

During the past decade, significant progress has been made in planning with actions that not only have discrete but also continuous effects. In the area of temporal and numeric planning, the category of problems where changes in the continuous state are affected by the durations of actions, is the most interesting to us, as it is the focus of Kongming. I describe the major planners that address such problems and compare them with Kongming in this section.

Zeno [PW94] is among the early work that explores planning with continuous change. It uses first-order logic for problem description and replies on constraint satisfaction to perform all temporal and metric reasoning during planning. In Zeno, the continuous processes have linear effects, and are described using differential equations. Although the continuous effects are linear both in Zeno and in Kongming, the differential equations of continuous processes in Zeno are first-order, while the state equations of continuous actions in Kongming can be nth-order. Zeno was impressively conceived as a first step towards automated planning with expressive metric temporal languages at the time. However, it is unable to solve problems that involve more than a dozen steps, and is limited to block-world type of planning.

Optop [McD03] is a heuristics-based planner that uses an estimated regression graph. It is reported that its performance is not promising due to the expensive computation of regression-match graphs.

TM-LPSAT [SD05] and the recent planner COLIN [CCFL09] can both construct plans in domains containing durative actions and linear continuous changes. They use PDDL+ and PDDL2.1 to represent the domain and problem. The differences between PDDL+/PDDL2.1 and Kongming's input were already described in Section 2.1. To recapitulate, TM-LPSAT and COLIN do not solve problems with polygonal continuous preconditions, coupled multiple-order action dynamics, external disjunctive linear constraints, and do not optimize with respect to an objective function, while Kongming solves such problems.

Another important advantage of Kongming is that, the linear continuous effects in the action specification of TM-LPSAT and COLIN are limited to a fixed rate of change. For example, a plane flies in a fixed direction at a constant speed (or a constant rate of change) of 250 m/s. This limitation prevents these planners from being applied to most autonomous vehicle applications, in which it is unreasonable to assume that the velocity (or acceleration) in the x, y and z directions is all constant. TM-LPSAT and COLIN both use a real-valued time line. Kongming uses a discrete-time framework to accommodate the discrete-time representation of the dynamics of continuous actions. The discrete time representation is commonly used in modeling dynamic systems in control [dV93]. Similar to Kongming, TM-LPSAT and COLIN both reason about durative actions with flexible temporal bounds.

TM-LPSAT encodes the PDDL+ problem description into LCNF [WW99], which is a propositional CNF formula with linear constraints, and uses the LPSAT [WW99] constraint solver to extract a feasible solution. Kongming first compactly represents all valid plans in a graph to reduce search space through mutual exclusion relations, then encodes the graph into a mixed logic quadratic program (MLQP), and uses an MLQP solver to extract an optimal solution. Because LPSAT outputs feasible solutions rather than optimal ones, TM-LPSAT cannot find an optimal plan with respect to an objective function. Kongming outputs an optimal plan according to any

user-defined linear or quadratic objective function.

COLIN is a forward state-space search planner that builds on a temporal planner CRIKEY3 [CFLS08b]. COLIN uses linear programs to represent temporal constraints and constraints on linear processes, and extends the temporal relaxed planning graph heuristic of CRIKEY3 to provide heuristic guidance in problems with continuous numeric effects. In contrast, Kongming uses a constraint-based planner to operate on its plan representation (Hybrid Flow Graph) and does not use heuristic search. A heuristic search planner may be employed on Kongming's plan representation, as discussed in future work (Chapter 9).

HAO* [MBB+09] is a heuristic search planner that handles uncertain action outcomes and uncertain continuous resource consumption. It formulates the planning problem as a hybrid-state Markov decision process (MDP), where the continuous state variables represent resources, such as battery power of a rover. HAO* uses heuristic search that is generalized from the AO* [Pea84] heuristic search algorithm to solve the MDP problem. The advantage of HAO* is that it allows uncertainty in the system and in the environment, which is closer to reality than Kongming's deterministic model. However, its MDP formulation can suffer from state space explosion due to the additional continuous state variables. An assumption of HAO* is that only resources are modeled as continuous variables. In contrast, Kongming reasons about not only continuous resources but also continuous trajectories of actions. HAO* also limits the resources to be monotonic, meaning that resources are consumed by actions but they cannot be replenished. This avoids revisiting states when solving the MDP, and hence, simplifies the problem.

2.3 Planning for Temporally Extended Goals

There is an extensive literature on planning to achieve temporally extended goals (TEGs). TLPlan [BK96] and TALPlan [KD00] treat TEGs as temporal domain control knowledge and prune the search space by progressing the temporal formula. [BCCZ99, LBHJ04, MR07] extend the planning as satisfiability approach [KS92].

[BM06] transforms the problem of planning for a TEG into a classical planning problem and applies heuristics to guide search. None of the existing TEG planners can plan with hybrid actions, where actions may have both discrete and continuous duration-dependent effects. To our knowledge, Kongming is the first to address these issues. Kongming plans for TEGs with both discrete and continuous actions, through encoding TEGs as durative actions and incorporating them in the plan representation.

TLPlan [BK96] uses a version of metric interval temporal logic (MITL) [AFH96], extended to allow first-order quantification to be expressed. Kongming uses a qualitative state plan (QSP) [Léa05, Chu08, Hof06] to represent a TEG. A QSP is defined and compared with MITL in Section 3.1.3. The QSP representation of TEGs in Kongming can be encoded as MITL formulae.

Kongming uses Flow Tubes [HW06, Hof06] to capture the continuous dynamics of actions, and builds upon the Planning Graph from Graphplan [BF97] and Blackbox [KS99]. I review them respectively in the following sections.

2.4 Flow Tubes

Flow tubes have been used in the qualitative reasoning community to represent a set of trajectories with common characteristics that connect two regions. Zhao [Zha92][3] uses them to characterize phase space during analysis of nonlinear dynamical systems. Hofmann [Hof06] uses flow tubes to represent bundles of state trajectories that take into account dynamic limitations due to under-actuation, while satisfying plan requirements for the foot placement of walking bipeds. By defining the valid operating regions for the state variables and control parameters in the abstracted model of a biped, the flow tubes prune infeasible trajectories and ensure plan execution. Similar representations are used in the robotics community, under the name funnels [Ted09], and in the hybrid systems community, under reachability analysis [KV07, KGBM04, Gir05, SK03, Kos01].

A large body of literature [CK98b, CK98a, AHH96, KGBM04, Gir05, SK03, Kos01, KV07] is dedicated to exact or approximate computation methods of flow tubes for

[3]They are called flow pipes in [Zha92].

representing sets of trajectories of a dynamic hybrid system from a given set of initial states. I compare them with the flow tube computation in Section 4.1.4.

The use of flow tubes in Kongming is inspired by Hofmann's work on executing qualitative state plans for continuous systems applied to controlling biped walking. [Hof06, HW06] use flow tubes to represent bundles of state trajectories of biped walking that satisfy plan requirements, while also satisfying dynamics and actuation limitations. The flow tubes are computed in order to project the feasible future evolution of the bipeds state, as shown in Fig. 2-3. Once the flow tubes are computed, a program executes the plan by adjusting control parameters in order to keep trajectories within the tubes.

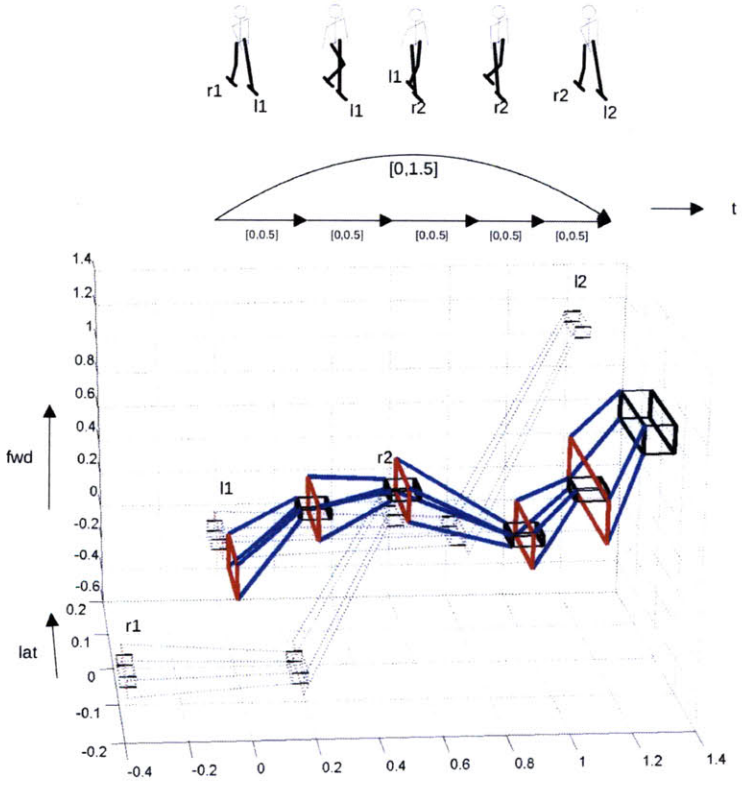


Figure 2-3: Flow tubes for biped center of mass are shown, with initial regions in red, goal regions in black, and tubes in blue. Flow tubes for left and right foot position are shown using dotted lines. As long as state trajectories remain within the flow tubes, the plan will execute successfully [Hof06].

There are two main reasons why Kongming uses flow tubes. Existing hybrid

planners that plan for both discrete and continuous actions are limited to first-order, decoupled state transition and a fixed rate of change, as described in Section 2.2. These limitations prevent those planners from being applied to most air/ground/underwater vehicle applications, in which it is unreasonable to assume that the velocity (or acceleration) in the x, y and z directions is all constant. Flow tubes can capture coupled nth-order linear or nonlinear dynamics of the actions, and provide an abstraction for representing and reasoning about the infinite number of trajectories of each action.

The second reason is for Kongming to have a least commitment strategy. When a hybrid action is performed, Kongming computes the feasible region in state space instead of a specific point through flow tube computation. This way Kongming provides not only temporal flexibility, but also spatial flexibility. Some of the existing hybrid planners provide temporal flexibility, but none of them is able to output spatially flexible plans.

For the afore mentioned reasons, Kongming uses flow tubes to represent the abstraction of an infinite number of trajectories and to compute the reach sets of hybrid actions.

2.5 Planning Graphs

In this section, I first review the Planning Graph in detail. This is because it is important to understand the Planning Graph in order to understand Kongming's plan representation, which I call the Hybrid Flow Graph.

The Planning Graph introduced in Graphplan [BF97], is a compact structure for representing the plan space in the STRIPS-like planning domain. It has been employed by a wide range of modern planners, including LPGP [LF02], STAN [LF99], GP-CSP [DK01a], to name a few. The Graphplan planner constructs and analyzes the planning graph structure, and extracts a valid plan from it. A Planning Graph represents all valid plans and prunes many invalid ones. Planning Graphs can be constructed quickly: they have polynomial size and can be built in polynomial time.

As shown in Fig. 4-7, a Planning Graph is a directed, leveled graph that alternates

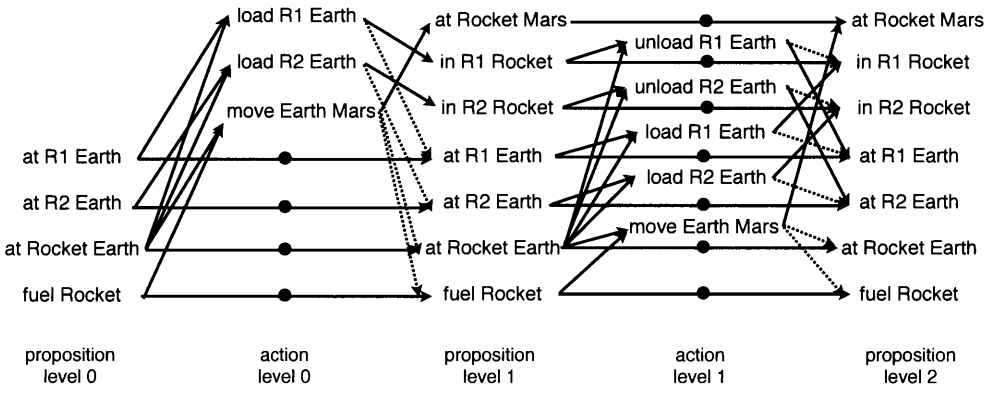


Figure 2-4: A Planning Graph consisting of 3 proposition levels and 2 action levels. Round black dots represent no-op actions. Solid lines connect preconditions with actions and actions with add-effects. Dotted lines connect actions with delete-effects.

between proposition levels, which contain propositions, and action levels, which contain actions. Proposition level i contains all propositions that can be true at time i, and action level i contains all possible actions to take place at time i.

Edges in a Planning Graph explicitly represent relations between actions and propositions. Propositions in proposition level i are connected to actions in action level i if the propositions are the preconditions of the actions. For example, in Fig. 4-7, `at` R_1 `Earth` and `at Rocket Earth` in proposition level 0 are connected to `load` R_1 `Earth` in action level 0, because `at` R_1 `Earth` and `at Rocket Earth` are the preconditions of action `load` R_1 `Earth`. Actions in action level i are connected to propositions in proposition level $i + 1$ if the propositions are the effects of the actions. For example, in Fig. 4-7, `load` R_1 `Earth` in action level 0 is connected to `in` R_1 `Rocket` and `at` R_1 `Earth` in proposition level 1, because `in` R_1 `Rocket` is the add-effect of `load` R_1 `Earth` and `at` R_1 `Earth` is the delete-effect of `load` R_1 `Earth`. Add-effects are connected using solid lines, and delete-effects are connected using dotted lines. Because no-op actions exist, every proposition that appears in proposition level i also appears in proposition level $i + 1$.

The mutual exclusion relations among proposition and action nodes make constraints inherent in the planning problem explicit. Two actions at a given action level are *mutually exclusive* if no valid plan can contain both. Similarly, two propositions

at a given proposition level are *mutually exclusive* if no valid plan can make both true. Identifying the mutual exclusions helps reduce the search for a subgraph of the Planning Graph that corresponds to a valid plan. The mutual exclusion relations are discovered and propagated through the Planning Graph using a few rules. They prove to be greatly helpful in pruning search space [BF97, KS99].

There are two main reasons why Kongming builds upon the Planning Graph framework. The first reason is to take advantage of the powerful mutual exclusion relations. Kongming defines a set of new rules such that continuous actions and continuous states are included in the mutual exclusion relations, which help reduce the hybrid plan space. The second reason is that Planning Graphs can be constructed quickly and easily: they have polynomial size and can be built in polynomial time.

2.6 Blackbox

Kongming encodes its plan representation, the Hybrid Flow Graph, into a constraint encoding, in an analogous way to the Blackbox planning system. Therefore, I review Blackbox in this section.

Blackbox [KS99] is a planning system that unifies the planning as satisfiability framework with the Planning Graph approach. Blackbox was inspired by a comparison between Graphplan and SATPLAN [KS92]. SATPLAN encodes the STRIPS input directly as a SAT problem and solves the SAT problem using a SAT solver. Graphplan is good at instantiating the propositional structure, while SATPLAN uses powerful search (extraction) algorithms [KS99]. Blackbox combines the advantages from both worlds: the instantiation power of Graphplan and the search capability from SAT solvers.

It encodes the Planning Graph as a SAT problem, and solves it using a state-of-the-art SAT solver. More specifically, Blackbox solves the planning problem in the following four steps:

- Turns STRIPS input into a Planning Graph;

- Translates the Planning Graph to a conjunctive normal formula in propositional logic;
- Applies simplification techniques, including unit propagation, failed literal and binary failed literal;
- Solves using a SAT solver.

Kongming takes an analogous approach. It combines the instantiation power of Graphplan, extended for hybrid plans, and the search capability from state-of-the-art constraint solvers. More specifically, Kongming encodes the representation for hybrid plans, called Hybrid Flow Graph, as a mixed logic quadratic program (MLQP), and solve the MLQP problem using a state-of-the-art MLQP solver.

To summarize, in this chapter I discussed related work in terms of the problem representation, planning for actions with continuous changes, and planning for temporally extended goals. I also described the three main pieces of prior work that Kongming builds upon: flow tubes, Graphplan, and Blackbox. In the next chapter, I formally define the problem statement.

Chapter 3

Problem Statement

Contents

In this thesis, I solve the problem of generating an optimal hybrid plan that achieves a temporally extended goal given the initial state of the system under control.

A temporally extended goal is a set of goals that are temporally constrained. The temporally extended goal for the underwater example in Section 1.2 is shown pictorially in Fig. 3-1. Model-based programming [WICE03] is a framework that allows engineers to program reactive systems by specifying desired behaviors in terms of qualitative descriptions of state evolution. This desired state evolution is called a qualitative state plan (QSP) [Chu08, Léa05, Hof06]. In this thesis, I use a QSP to represent a temporally extended goal. I define a QSP in Def. 5 later on in this chapter.

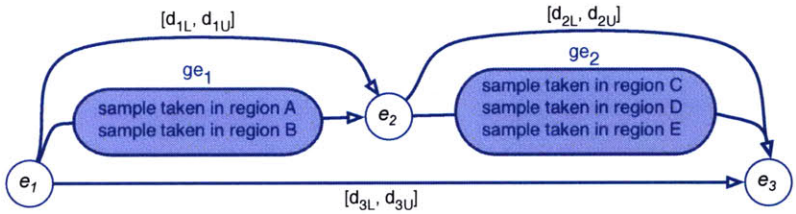


Figure 3-1: A temporally extended goal example for the underwater scenario associated with Fig. 1-1.

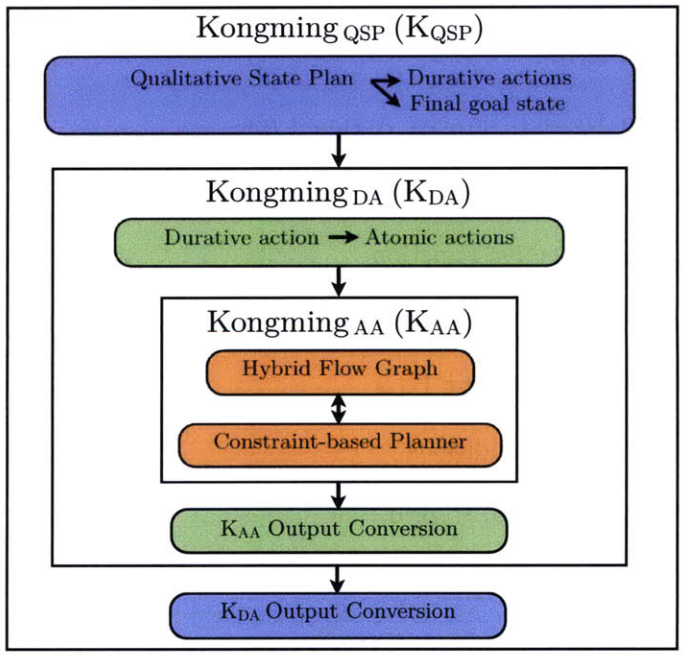


Figure 3-2: Overview of Kongming's approach.

The problem statement is divided into three parts, corresponding to the three planners in Kongming, as shown in Fig. 3-2. As described in Chapter 1, Kongming$_{QSP}$

or K_{QSP} is a planner that plans for QSPs instead of a final goal state, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. Kongming$_{DA}$ or K_{DA} is a planner that handles durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. Kongming$_{AA}$ or K_{AA} is a planner that handles atomic actions and a final goal state.

I define the problem for each planner in the following sections.

3.1 Problem Statement for K_{QSP}

The planning problem K_{QSP} solves is, given *initial conditions*, a *qualitative state plan*, *hybrid durative action types*, *external constraints*, and an *objective function*, find an optimal sequence of actions from the given action types and an optimal state trajectory that achieve the the QSP without violating any of the constraints.

First, I formally define the hybrid planning problem as follows:

Definition 1. [Hybrid QSP Planning Problem] *Given* $< \mathbf{s}, \mathbf{u}, \mathcal{I}, \mathcal{QSP}, \mathcal{DA}, \mathcal{C}, f >$, *a* ***hybrid QSP planning problem*** *is to return* $\mathcal{P}_{qsp}$, *where*

- $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$ *is a set of real-valued and propositional state variables (Def. 2),*
- $\mathbf{u}$ *is a set of real-valued control variables (Def. 3),*
- $\mathcal{I}$ *is the initial conditions (Def. 4),*
- $\mathcal{QSP}$ *is a qualitative state plan (Def. 5),*
- $\mathcal{DA}$ *is a set of hybrid durative action types (Def. 10),*
- $\mathcal{C}$ *is a set of external constraints (Def. 16),*
- f *is an objective function (Def. 17),*
- $\mathcal{P}_{qsp}$ *is a hybrid optimal plan (Def. 18).*

Next, I define each of them in detail. In the following subsections, I demonstrate these concepts using the example of an unmanned air vehicle performing a fire fighting

mission. The map for the scenario is shown in Fig. 3-3. To simplify the example, the world is assumed to be 2D and the air vehicle is assumed to fly at a constant altitude. There are two forest fires that need to be extinguished. Fire 1 has a higher priority because it is closer to a residential area. There are two no fly zones (NFZs) that the vehicle must avoid and two lakes. There is an unmanned air vehicle at the base that is equipped with a water tank and a camera. It can take water from the lakes, put out fires, and take photos.

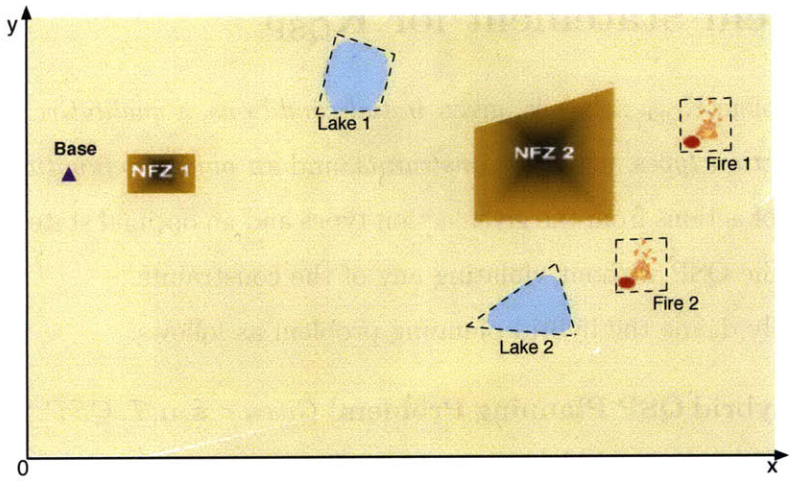


Figure 3-3: The map of the fire fighting scenario. There are two forest fires that need to be extinguished. Fire 1 has a higher priority because it is closer to a residential area. There are two no fly zones (NFZs) and two lakes.

3.1.1 Input: Variables

The state variables, $\mathbf{s}$, represent the continuous and discrete state of the system.

Definition 2. [State Variables] $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$, *where* $\mathbf{x} \in \Re^n$ *is a vector of real-valued variables,* $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$, *and* $\mathbf{p}$ *is a vector of propositional variables,* $\mathbf{p} = \langle p_1, p_2, \ldots, p_l \rangle$.

In the fire fighting example, the real-valued state variables are the position, velocity and fuel of the air vehicle, $\mathbf{x} = \langle x, y, v_x, v_y, fuel \rangle$, and the propositional state variables are $\mathbf{p} = \langle$*fire1-out, fire2-out, have-water, photo1-taken, photo2-taken*$\rangle$.

The control variables, $\mathbf{u}$, represent the control input to the system.

Definition 3. [Control Variables] $\mathbf{u} \in \Re^m$ *is a vector of real-valued control variables,* $\mathbf{u} = \langle u_1, u_2, \ldots, u_m \rangle$.

In the fire fighting example, the control variables are the acceleration and the fuel consumption rate of the air vehicle, $\mathbf{u} = \langle a_x, a_y, r_f \rangle$.

3.1.2 Input: Initial Conditions

The initial conditions, $\mathcal{I}$, specify constraints on the initial value of the state variables $\mathbf{s}$.

Definition 4. [Initial Conditions] $\mathcal{I} = (\mathbf{Ax} \leq b) \wedge_i l_i$, *where* $\mathbf{Ax} \leq b$ *is a system of linear inequalities over the real-valued state variables* $\mathbf{x}$, *and each* l_i *is a positive or negative literal for a propositional state variable* $p_i \in \mathbf{p}$.

In the fire fighting example, the initial conditions constrain the initial position, velocity and fuel of the air vehicle, and specify the initial value of the propositional state variables. $\mathcal{I} = \{2x - y \geq -38, 2x + y \leq 68, y \geq 50, v_x = 0, v_y = 0, fuel = 100,$ *¬fire1-out, ¬fire2-out, ¬have-water, ¬photo1-taken, ¬photo2-taken*$\}$. This translates to that, the air vehicle is initially at base, with zero velocity and 100 units of fuel; neither fire is extinguished; the air vehicle has no water; and no photo has been taken.

3.1.3 Input: QSP

In model-based programming, a goal specification formalism called a qualitative state plan (QSP) is used to describe a temporally extended goal. In effect, a temporally extended goal specifies the allowed state trajectories of the system.

Similar to the definition of a QSP in [Chu08], I define it as follows.

Definition 5. [QSP] *A qualitative state plan (QSP) is a 3-tuple* $\langle \mathcal{E}, \mathcal{GE}, \mathcal{C}_\mathcal{T} \rangle$, *where*

- $\mathcal{E} = \{e_1, \ldots, e_m\}$ *is a finite set of events, representing points in time (Def. 6).*
- $\mathcal{GE} = \{ge_1, \ldots, ge_n\}$ *is a finite set of goal episodes that specify the desired state evolution over time (Def. 7).*

- $\mathcal{C}_\mathcal{T}$ *is a finite set of temporal constraints on events* $\mathcal{E}$ *(Def. 9).*

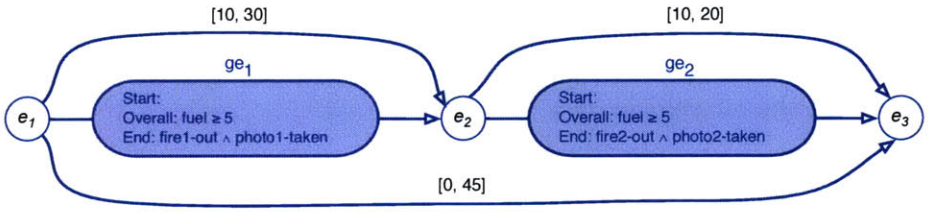


Figure 3-4: A QSP example of the fire fighting scenario. There are three events (time points), e_1, e_2 and e_3. There are two goal episodes, ge_1 and ge_2. There are three temporal constraints, one between e_1 and e_2, one between e_2 and e_3, and one between e_1 and e_3.

I illustrate a QSP diagrammatically by an acyclic directed graph in which events are represented by nodes, goal episodes by arcs labeled with associated state constraints, and temporal constraints by arcs. The QSP of the fire fighting example is shown in Fig. 3-4.

Event

Definition 6. [Event of a QSP] *An event of a QSP is a real-valued time point,* $e \in \Re$. *The time for each* e_i *is measured from a reference event called the* ***QSP start event****, and denoted as* e_s.

In the QSP shown in Fig. 3-4, there is a total of three events, $\mathcal{E} = \{e_1, e_2, e_3\}$. e_1 is the QSP start event, that is $e_s \equiv e_1$.

Goal Episode

Each goal episode in a QSP specifies the desired state of the system under control over a sub-interval of the planning horizon. Each goal episode is associated with state constraints, a start event and an end event.

Definition 7. [Goal Episode of a QSP] *A goal episode, ge, is a 3-tuple* $\langle e_i, e_j, C_s \rangle$, *where*

- $e_i \in \mathcal{E}$ *is the start event of the goal episode.*

- $e_j \in \mathcal{E}$ *is the end event of the goal episode, such that* $e_i \leq e_j$.
- C_s *is a state constraint over the state variables (Def. 8).*

In the QSP shown in Fig. 3-4, there is a total of two goal episodes, $\mathcal{GE} = \{ge_1, ge_2\}$.

Definition 8. [**State Constraint of a Goal Episode**] *A state constraint of a goal episode,* C_s*, is a 3-tuple* $\langle c_s, c_o, c_e \rangle$*, where*

- c_s *is the* ***start*** *state constraint, representing the constraint that holds at the start event of the goal episode.*
- c_o *is the* ***overall*** *state constraint, representing the constraint that holds over the duration of the goal episode, excluding the start event and the end event.*
- c_e *is the* ***end*** *state constraint, representing the constraint that holds at the end event of the goal episode.*

c_s*,* c_o *and* c_e *are all specified by* $(\mathbf{Ax} \leq b) \wedge_i l_i$*, where* $\mathbf{Ax} \leq b$ *is a system of linear inequalities over the real-valued state variables* $\mathbf{x}$*, and* l_i *is a positive or negative literal for a propositional state variable* $p_i \in \mathbf{p}$.

In the QSP shown in Fig. 3-4, in both goal episodes, ge_1 and ge_2, there is no *start* state constraint, and the *overall* state constraint is $fuel \geq 5$. The *end* state constraint of ge_1 is *fire1-out* $\wedge$ *photo1-taken*. The *end* state constraint of ge_2 is *fire2-out* $\wedge$ *photo2-taken*. Goal episode ge_1 requires that, at the end of the episode, Fire 1 is put out and a photo is taken, and throughout the episode, there is at least 5 units of fuel on the air vehicle. Similar for goal episode ge_2. Moreover, Fire 1 is required to be put out before Fire 2.

Temporal Constraint

The last element of a QSP is temporal constraints. They specify temporal bounds between two events.

Definition 9. [**Temporal Constraint of a QSP**] *A temporal constraint,* $\mathcal{C}_\mathcal{T}$*, is a simple temporal constraint (STC) [DMP91].* $\mathcal{C}_\mathcal{T} = \langle e_i, e_j, lb, ub \rangle$*, where*

- $e_i \in \mathcal{E}$ *is the starting event of the temporal constraint.*
- $e_j \in \mathcal{E}$ *is the end event of the temporal constraint, such that* $e_i < e_j$.
- $lb \in \Re \cup \{-\infty\}$ *is the lower bound on the time between* e_i *and* e_j.
- $ub \in \Re \cup \{+\infty\}$ *is the upper bound on the time between* e_i *and* e_j.

In the QSP shown in Fig. 3-4, there are three temporal constraints. One requires the time between e_1 and e_2 to be at least 10 units and at most 30 units. One requires the time between e_2 and e_3 to be at least 10 units and at most 20 units. One requires the time between e_1 and e_3 to be at least 0 units and at most 45 units.

Comparison of QSP with MITL

I compare my definitions of a QSP to Metric Interval Temporal Logic (MITL) introduced in [AFH96]. MITL is a logic-based language introduced to specify desired behaviors of real-time systems. Like a QSP, it uses a dense representation of time. An MITL formula is formally defined in Eq. 3.1, where p is a proposition that is required to be true. $\mathcal{U}_I$ is the *until* operator, where I is a time interval with left bound $l(I)$ and right bound $r(I)$. By definition, $\phi_1 \; \mathcal{U}_I \; \phi_2$ is true at time t if and only if ϕ_1 is true at time t, and remains true for at least $l(I)$ and at most $r(I)$ time units, *until* ϕ_2 becomes true.

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U}_I \phi_2 \tag{3.1}$$

Additional operators can be constructed from Eq. 3.1, including the *eventually* operator $\Diamond_I$, the *always* operator $\Box_I$, and the *unless* operator ${}_I\mathcal{W}$. $\Diamond_I \phi$ means that ϕ must *eventually* be true at some time instant within the time interval I, and is equivalent to $\texttt{true} \; \mathcal{U}_I \; \phi$. $\Box_I \phi$ means that ϕ must *always* be true during the time interval I, and is equivalent to $\neg\Diamond_I\neg\phi$. Finally, $\phi_1 \; {}_I\mathcal{W} \; \phi_2$ means that ϕ_1 must be true at all times during the time interval I, *unless* ϕ_2 becomes true before the end of interval I, in which case ϕ_1 may only be true until ϕ_2 becomes true. $\phi_1 \; {}_I\mathcal{W} \; \phi_2$ is equivalent to $\neg((\neg\phi_2) \; \mathcal{U}_I \; (\neg\phi_1))$.

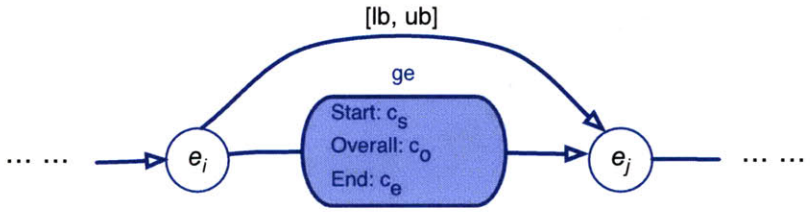


Figure 3-5: Part of a QSP. There is a goal episode ge and a temporal constraint c_T between two events e_i and e_j.

I next show that any QSP can be expressed in MITL. Suppose between two events e_i and e_j in a QSP, there is a goal episode $ge = \langle e_i, e_j, c_s, c_o, c_e \rangle$ and a temporal constraint $c_T : [lb, ub]$, where c_s, c_o and c_e are the *start*, *overall* and *end* state constraints for the goal episode, and lb, ub are the lower and upper bounds of the temporal constraint, as shown in Fig. 3-5. The goal episode and temporal constraint can be expressed in MITL as follows.

- Let interval I be $[lb, ub]$, and hence, the left bound $l(I) = lb$ and the right bound $r(I) = ub$;
- $c_s \wedge (c_o \, \mathcal{U}_I \, c_e)$ holds at time e_i;
- c_e holds at time e_j.

Thus, it is enforced that the *start* state constraint c_s holds at the start event of the goal episode, the *end* state constraint c_e holds at the end event of the goal episode, and the *overall* state constraint c_o holds in between for at least lb and at most ub.

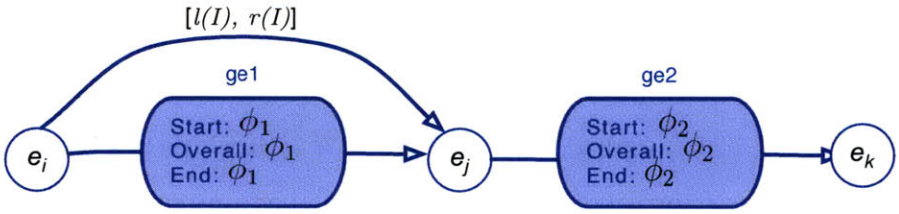


Figure 3-6: A QSP representing $\phi_1 \mathcal{U}_I \phi_2$. Goal episode $ge1$ specifies that ϕ_1 is true for at least $l(I)$ and at most $r(I)$. Goal episode $ge2$ specifies that ϕ_2 is true immediately after.

On the other hand, any non-disjunctive MITL can be expressed in QSP. We focus on the until operator, $\phi_1 \mathcal{U}_I \phi_2$, to explain the mapping. Recall that $\phi_1 \, \mathcal{U}_I \, \phi_2$ is true

at time t if and only if ϕ_1 is true at time t, and remains true for at least $l(I)$ and at most $r(I)$ time units, *until* ϕ_2 becomes true. This can be expressed in QSP as in Fig. 3-6.

3.1.4 Input: Hybrid Durative Action Types

$\mathcal{DA}$ specifies the set of hybrid durative action types that a system can perform. The actions are hybrid, as they have both continuous and discrete effects. The actions are durative, as they have flexible durations.

Definition 10. [Hybrid Durative Action Type] *A hybrid durative action type is a 4-tuple,* $\langle Cond, Eff, Dyn, d \rangle$*, where*

- $Cond$ *is a finite set of* ***conditions*** *of the hybrid durative action type (Def. 11).*
- Eff *is a finite set of* ***discrete effects*** *of the hybrid durative action type (Def. 12).*
- Dyn *is the* ***dynamics*** *of the hybrid durative action type (Def. 13).*
- $d \in \Re$ *is the* ***duration*** *of the hybrid durative action type (Def. 15).*

Similar to the PDDL languages [MtAPCC98, FL03, FL01], the action types are parameterized, such that one action type can be applied to various objects by various subjects.

As an example, in the fire fighting scenario, the air vehicle can perform the following hybrid durative action types: `fly`, `get-water`, `extinguish-fire`, `take-photo`. The action specifications are shown in Fig. 3-7.

Action `get-water` can be applied to both lakes. "?l - `lake`" means that l is of the type `lake`. Each lake l corresponds to a specific polyhedron region on the map (Fig. 3-3): $lake1 = \{3x - y \geq 64, x + 2y \leq 180, x - 4y \leq -184, 7x - 2y \leq 220\}$, and $lake2 = \{7x - 11y \geq 167, 5x + 2y \leq 415, x + 5y \geq 175\}$. Likewise, action `extinguish-fire` and `take-photo` can be applied to both fire regions. Each fire region f corresponds to a specific polyhedron region on the map (Fig. 3-3): $fire1 = \{x \leq 95, x \geq 90, y \leq 58, y \geq 52\}$, and $fire2 = \{x \leq 85, x \geq 80, y \leq 32, y \geq 26\}$.

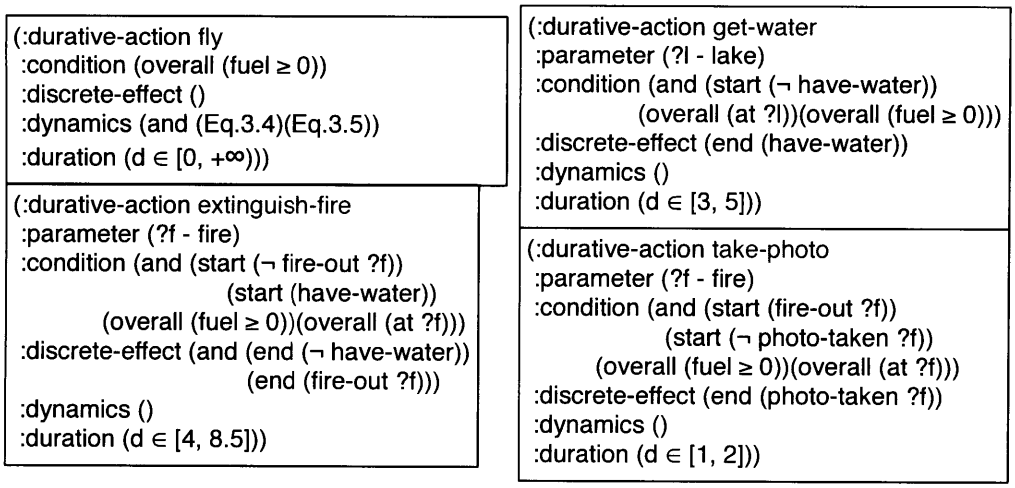


Figure 3-7: In the fire fighting scenario, the air vehicle can perform the following hybrid durative action types: `fly`, `get-water`, `extinguish-fire`, `take-photo`.

Conditions

Conditions are requirements that must be satisfied in order for an action to take place. They are specified in different stages of the action: *start*, *overall*, and *end*.

Definition 11. [Condition of a Hybrid Durative Action Type] *A condition of a hybrid durative action type is a 3-tuple,* $\langle cond_s, cond_o, cond_e \rangle$*, where*

- $cond_s$ *is the* ***start*** *condition, representing the state constraint that holds at the start point of the action.*
- $cond_o$ *is the* ***overall*** *condition, representing the state constraint that holds for the duration of the action, excluding the start and the end point of the action.*
- $cond_e$ *is the* ***end*** *condition, representing the state constraint that holds at the end point of the action.*

$cond_s$*,* $cond_o$ *and* $cond_e$ *are all specified by* $(\mathbf{Ax} \leq b) \wedge_i l_i$*, where* $\mathbf{Ax} \leq b$*, called the continuous condition, is a system of linear inequalities over the real-valued state variables* $\mathbf{x}$*, and each* l_i*, called a discrete condition, is a positive or negative literal for a propositional state variable* $p_i \in \mathbf{p}$*.*

In the fire fighting scenario, as shown in Fig. 3-7, the condition of action `fly` is that the vehicle has fuel throughout the action; the condition of action `get-water` is that the vehicle has no water at the start, is at the lake throughout, and has fuel throughout the action; the condition of action `extinguish-fire` is that the fire is not out at the start, the vehicle has water at the start, has fuel throughout, and is at the fire throughout the action; and the condition of action `take-photo` is that the fire is out at the start, photo of the fire is not taken at the start, the vehicle has fuel throughout, and is at the fire throughout the action. Note that the condition of being at lake or being at fire corresponds to a system of linear inequalities over the state variables. For example, the condition of being at Lake 1 translates to Eq. 3.2.

$$(x, y) \in \{3x - y \geq 64, x + 2y \leq 180, x - 4y \leq -184, 7x - 2y \leq 220\} \tag{3.2}$$

Discrete Effects

Discrete effects capture the changed propositional truth values as a result of the action. They are specified in different stages of the action: *start*, *overall*, and *end*.

Definition 12. [Discrete Effect of a Hybrid Durative Action Type] *A discrete effect of a hybrid durative action type is a 3-tuple, $\langle eff_s, eff_o, eff_e \rangle$, where*

- *eff_s is the **start** effect, which is true at the start point of the action.*
- *eff_o is the **overall** effect, which is true for overall of the action.*
- *eff_e is the **end** effect, which is true at the end point of the action.*

eff_s, eff_o and eff_e are all specified by $\wedge_i l_i$, where l_i is a positive or negative literal of a propositional variable $p_i \in \mathbf{p}$.

In the fire fighting scenario, as shown in Fig. 3-7, the discrete effect of action `fly` is unspecified; the discrete effect of action `get-water` is that the vehicle has water at the end of the action; the discrete effect of action `extinguish-fire` is that the fire

is out at the end, and the vehicle has no water at the end of the action; the discrete effect of action `take-photo` is that photo of the fire is taken at the end of the action.

Dynamics

Dynamics describe the state transition and the actuation limits of a system when performing an action.

Definition 13. [Dynamics of a Hybrid Durative Action Type] *Dynamics of a hybrid durative action type is a pair, $\langle Tran_S, Lim \rangle$, where*

- *$Tran_S$ represents the state transition equation (Def. 14).*
- *Lim, representing the actuation limits, is a pair, $\langle \mathbf{lb}, \mathbf{ub} \rangle$, where $\mathbf{lb} \in \Re^m$ and $\mathbf{ub} \in \Re^m$ are lower and upper bounds on the control variables $\mathbf{u}$: $\mathbf{u} \in [\mathbf{lb}, \mathbf{ub}]$.*

Definition 14. [State Equation of a Hybrid Durative Action Type] *Given a time discretization $\langle t_0, t_1, \ldots \rangle \in \Re^{\mathcal{N}}$, a* **state equation** *is a linear relation, expressing the value of the real-valued state variables $\mathbf{x}$, at all time steps t_i, as a function of $\mathbf{x}$ and $\mathbf{u}$, at time step t_{i-1}, in the form of Eq. 3.3,*

$$\forall t_i,\ \mathbf{x}(t_i) = \mathbf{A}\mathbf{x}(t_{i-1}) + \mathbf{B}\mathbf{u}(t_{i-1}), \tag{3.3}$$

where $\mathbf{A}$ is a $n \times n$ matrix of constants, and $\mathbf{B}$ is a $n \times m$ matrix of constants

In the fire fighting scenario, as shown in Fig. 3-7, the dynamics of action `fly` are shown in Eq. 3.4 and Eq. 3.5 ; and the dynamics of other actions are unspecified.

$$\begin{bmatrix} x \\ y \\ v_x \\ v_y \\ fuel \end{bmatrix}(t_i) = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ v_x \\ v_y \\ fuel \end{bmatrix}(t_{i-1}) + \begin{bmatrix} \frac{\Delta t^2}{2} & 0 & 0 \\ 0 & \frac{\Delta t^2}{2} & 0 \\ \Delta t & 0 & 0 \\ 0 & \Delta t & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \times \begin{bmatrix} a_x \\ a_y \\ r_f \end{bmatrix}(t_{i-1}) \tag{3.4}$$

$$
\begin{aligned}
a_x &\in [-7.5, 7.5] \\
a_y &\in [-7.5, 7.5] \\
r_f &\in [-1, -1]
\end{aligned} \tag{3.5}
$$

Eq. 3.4 shows how the continuous state of the air vehicle (position, velocity and fuel) evolves from time t_{i-1} to time t_i. As shown in Eq. 3.5, the fuel consumption rate is -1, while the x and y accelerations of the air vehicle can take any value within the range specified in Eq. 3.5.

Duration

Actions have flexible durations. Each duration is a real-valued variable bounded by a lower bound and an upper bound.

Definition 15. [Duration of a Hybrid Durative Action Type] *The duration of a hybrid durative action type, d, is a pair, $\langle d_l, d_u \rangle$, where $d_l \in \Re$ is the lower bound and $d_u \in \Re$ is the upper bound, such that $d \in [d_l, d_u]$.*

In the fire fighting scenario, the bounds on the duration for each action is shown in Fig. 3-7.

3.1.5 Input: External Constraint

External constraints, $\mathcal{C}$, impose external requirements on the state of the system under control. They are in conjunctive normal form (CNF), and are specified by a conjunction of clauses, each of which is a disjunction of linear inequalities over the state variables $\mathbf{x}$.

Definition 16. [External Constraint] *An external constraint c is defined as a conjunction of disjunctions of linear inequalities, $\wedge_i(\vee_j\ a'_{ij}\mathbf{x} \leq b_{ij})$, where $\mathbf{x} \in \Re^n$ is the continuous state variables, a'_{ij} is the transpose of a_{ij}, $a_{ij} \in \Re^n$ is a vector of constants, and $b_{ij} \in \Re$ is a constant.*

Each disjunction, $\vee\ a'\mathbf{x} \leq b$, is called a clause. When a clause is composed only of one linear inequality, it is called a unit clause.

In the fire fighting scenario, the external constraints require that the position of the air vehicle be outside the no fly zones (NFZs) and be inside the map at all times. The constraint for outside NFZ 1 is shown in Eq. 3.6. The constraint for outside NFZ 2 is shown in Eq. 3.7. The constraint for inside the map is shown in Eq. 3.8, which is a conjunction of unit clauses.

$$\forall t,\ x \geq 26 \vee x \leq 18 \vee y \geq 54 \vee y \leq 49 \tag{3.6}$$

$$\forall t,\ x \geq 80 \vee x \leq 63 \vee y \leq 34 \vee 5x - 17y \leq -620 \tag{3.7}$$

$$\forall t,\ \{x \geq 0, x \leq 100, y \geq 0, y \leq 70\} \tag{3.8}$$

3.1.6 Input: Objective Function

An objective function f is a linear or quadratic function to be minimized. It is a function over real-valued state variables $\mathbf{x}$, control variables $\mathbf{u}$, and time. In this thesis, the objective function is limited to be linear or quadratic, purely due to the limitation of the state-of-the-art constraint solver that is used.

Definition 17. [Objective Function] *An objective function is a function from* $\Re^{n+m+1}$ *to* $\Re$, $f : \Re^{n+m+1} \mapsto \Re$. *It is a linear or quadratic function,* $f = a'\mathbf{X}$ *or* $f = \mathbf{X}'A\mathbf{X} + b'\mathbf{X}$, *where* $\mathbf{X} = [\mathbf{x}\ \mathbf{u}\ t]'$ *is the combined vector of the state variables, control variables and time;* a' *is the transpose of* a, $a \in \Re^{n+m+1}$ *is a vector of constant;* $A : \Re^{n+m+1} \times \Re^{n+m+1}$ *is a constant matrix;* b' *is the transpose of* b, *and* $b \in \Re^{n+m+1}$ *is a vector of constants.*

In the fire fighting scenario, typical objective functions are to minimize fuel use, distance traveled, or mission completion time.

3.1.7 Output: Optimal Hybrid Plan

To recapitulate, the hybrid planning problem for $\mathrm{K_{QSP}}$ is, to output an optimal hybrid plan $\mathcal{P}_{QSP}$, given an initial state, a QSP, a set of available action types, external constraints to satisfy and an objective function. I define the optimal hybrid plan in this section.

Definition 18. [Optimal Hybrid Plan of $\mathbf{K_{QSP}}$] *An optimal hybrid plan of K_{QSP} is a triple, $\mathcal{P}_{QSP} =< \mathbf{A}^*, \mathbf{S}^*, \mathbf{U}^* >$, where*

- $\mathbf{A}^* = \{\mathbf{a_d^*}(t_{n_1}), \mathbf{a_d^*}(t_{n_2}), \ldots, \mathbf{a_d^*}(t_{n_m})\}$ *is an optimal sequence of action-duration pairs; $\mathbf{a_d^*}(t_{n_i})$ is the set of $\langle a, d \rangle$ pairs at time t_{n_i}; in each pair, a is an action instantiated from the hybrid durative action types $\mathcal{DA}$ that starts at time t_{n_i}, and d is its duration;*

- $\mathbf{S}^* = \{\mathbf{s}^*(t_1), \mathbf{s}^*(t_2), \ldots, \mathbf{s}^*(t_N)\}$ *is an optimal sequence of assignment to the continuous and discrete state variables $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$, $\mathbf{s}^*(t_i)$ is the assignment to $\mathbf{s}$ at time t_i;*

- $\mathbf{U}^* = \{\mathbf{u}^*(t_1), \mathbf{u}^*(t_2), \ldots, \mathbf{u}^*(t_{N-1})\}$ *is an optimal sequence of assignment to the control variables $\mathbf{u}$, $\mathbf{u}^*(t_i)$ is the assignment to $\mathbf{u}$ at time t_i;*

such that all of the following are true:

- $T' \subseteq T$, *where* $T' = \{t_{n_1}, t_{n_2}, \ldots, t_{n_m}\}$ *and* $T = \{t_1, t_2, \ldots, t_N\}$, *and* $\forall t_i, t_{i-1} \in T$, $t_i - t_{i-1} = \Delta t$, *where Δt is a discretization constant.*

- *Initial conditions are satisfied.* $\mathbf{s}^*(t_1) = \langle \mathbf{x}^*(t_1), \mathbf{p}^*(t_1) \rangle \in \mathcal{I}$, *where $\mathcal{I}$ is the initial conditions;*

- *QSP is satisfied.* $\forall ge_i, C_T^j \in \mathcal{QSP}$, $\mathbf{S}^*$ *satisfy C_s^i and C_T^j, where $\mathcal{QSP}$ is the input QSP, ge_i is a goal episode of the QSP, C_s^i is the state constraint of the goal episode, and C_T^j is a temporal constraint of the QSP;*

- *No actions that overlap in time are mutex.* $\forall t_i \in T$, *no actions that take place at time t_i are mutex (Table 4.1);*

- *If an action is performed at time t_i, then all its preconditions are resolved (Def. 33) at time t_i;*

- *If a fact is true at time t_i, where $i > 1$, then at least one action that causes the fact to be true is performed at time t_{i-1};*

- *Each action satisfies its dynamics. $\forall t_i = t_1, \ldots, t_{N-1}$, $\mathbf{u}^*(t_i) \in Lim(a(t_i))$, $\forall a$ that takes place at time t_i, and $\{\mathbf{x}^*(t_i), \mathbf{u}^*(t_i), \mathbf{x}^*(t_{i+1})\}$ satisfy $Tran_S(a(t_i))$, $\forall a$ that takes place at time t_i, where $Lim(a(t_i))$ is the dynamic limitation of action a that takes place at time t_i, and $Tran_S(a(t_i))$ is the state transition equation of action a that takes place at time t_i;*

- *The duration of each action satisfies the duration bounds of its action type. $\forall t_{n_i} \in T'$, $\forall \langle a, d \rangle \in \mathbf{a}^*_{\mathbf{d}}(t_{n_i})$, $d \in [d_l, d_u]$, where $[d_l, d_u]$ are the duration bounds of action a;*

- *External constraints are satisfied. $\forall c_i \in \mathcal{C}$, the optimal continuous state variables $\{\mathbf{x}^*(t_1), \mathbf{x}^*(t_2), \ldots, \mathbf{x}^*(t_N)\}$ satisfy c_i, where $\mathcal{C}$ is the external constraints;*

- *The objective function f is minimized.*

3.2 Problem Statement for $\mathrm{K_{DA}}$

The previous section defined the problem that $\mathrm{K_{QSP}}$ solves. In this section, I define the problem that $\mathrm{K_{DA}}$ solves. Recall that $\mathrm{K_{QSP}}$ reformulates the QSP to durative actions and a final goal state, and then engages $\mathrm{K_{DA}}$. $\mathrm{K_{DA}}$ is a planner that handles durative actions with flexible durations.

Definition 19. [Hybrid Durative Action Planning Problem] *Given $< \mathbf{s}, \mathbf{u}, \mathcal{I}, \mathcal{G}, \mathcal{DA}, \mathcal{C}, f >$, the hybrid durative action planning problem of K_{DA} is to return $\mathcal{P}_{da}$, where*

- $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$ *is a set of real-valued and propositional state variables (Def. 2),*

- $\mathbf{u}$ *is a set of real-valued control variables (Def. 3),*

- $\mathcal{I}$ *is the initial conditions (Def. 4),*
- $\mathcal{G}$ *is the final goal conditions (Def. 20),*
- $\mathcal{DA}$ *is a set of hybrid durative action types (Def. 10),*
- $\mathcal{C}$ *is a set of external constraints (Def. 16),*
- f *is an objective function (Def. 17),*
- $\mathcal{P}_{da}$ *is a hybrid optimal plan (Def. 21).*

Def. 19 is similar to the hybrid QSP planning problem (Def. 1) that $\mathrm{K_{QSP}}$ solves, with two differences. First, the goals in Def. 1 are expressed by a QSP, whereas the goals in Def. 19 are expressed as the final goal conditions, $\mathcal{G}$. Second, the output hybrid optimal plan is slightly different.

3.2.1 Input: Goal Conditions

The goal conditions, $\mathcal{G}$, specify constraints on the final value of the state variables $\mathbf{s}$.

Definition 20. [Goal Conditions] $\mathcal{G} = (\mathbf{Ax} \leq b) \wedge_i l_i$, *where* $\mathbf{Ax} \leq b$ *is a system of linear inequalities over the real-valued state variables* $\mathbf{x}$, *and each* l_i *is a positive or negative literal for a propositional state variable* $p_i \in \mathbf{p}$.

In the fire fighting example, the goal conditions constrain the final position, velocity and fuel of the air vehicle, and specify the final value of the propositional state variables.

3.2.2 Output: Optimal Hybrid Plan

Definition 21. [Optimal Hybrid Plan of $\mathbf{K_{DA}}$] *An optimal hybrid plan of* K_{DA} *is a triple,* $\mathcal{P}_{DA} =< \mathbf{A}^*, \mathbf{S}^*, \mathbf{U}^* >$, *where*

- $\mathbf{A}^* = \{\mathbf{a_d^*}(t_{n_1}), \mathbf{a_d^*}(t_{n_2}), \ldots, \mathbf{a_d^*}(t_{n_m})\}$ *is an optimal sequence of action-duration pairs;* $\mathbf{a_d^*}(t_{n_i})$ *is the set of* $\langle a, d \rangle$ *pairs at time* t_{n_i}*; in each pair,* a *is an action instantiated from the hybrid durative action types* $\mathcal{DA}$ *that starts at time* t_{n_i}, *and* d *is its duration;*

- $\mathbf{S}^* = \{\mathbf{s}^*(t_1), \mathbf{s}^*(t_2), \ldots, \mathbf{s}^*(t_N)\}$ *is an optimal sequence of assignment to the continuous and discrete state variables* $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$, $\mathbf{s}^*(t_i)$ *is the assignment to* $\mathbf{s}$ *at time* t_i*;*

- $\mathbf{U}^* = \{\mathbf{u}^*(t_1), \mathbf{u}^*(t_2), \ldots, \mathbf{u}^*(t_{N-1})\}$ *is an optimal sequence of assignment to the control variables* $\mathbf{u}$, $\mathbf{u}^*(t_i)$ *is the assignment to* $\mathbf{u}$ *at time* t_i*;*

such that all of the following are true:

- $T' \subseteq T$, *where* $T' = \{t_{n_1}, t_{n_2}, \ldots, t_{n_m}\}$ *and* $T = \{t_1, t_2, \ldots, t_N\}$, *and* $\forall t_i, t_{i-1} \in T$, $t_i - t_{i-1} = \Delta t$, *where* Δt *is a discretization constant.*

- *Initial conditions are satisfied.* $\mathbf{s}^*(t_1) = \langle \mathbf{x}^*(t_1), \mathbf{p}^*(t_1) \rangle \in \mathcal{I}$, *where* $\mathcal{I}$ *is the initial conditions;*

- *Goal conditions are satisfied.* $\mathbf{s}^*(N) = \langle \mathbf{x}^*(N), \mathbf{p}^*(N) \rangle \in \mathcal{G}$, *where* $\mathcal{G}$ *is the goal conditions;*

- *No actions that overlap in time are mutex.* $\forall t_i \in T$, *no actions that take place at time* t_i *are mutex (Table 4.1);*

- *If an action is performed at time* t_i, *then all its preconditions are resolved (Def. 33) at time* t_i*;*

- *If a fact is true at time* t_i, *where* $i > 1$, *then at least one action that causes the fact to be true is performed at time* t_{i-1}*;*

- *Each action satisfies its dynamics.* $\forall t_i = t_1, \ldots, t_{N-1}$, $\mathbf{u}^*(t_i) \in Lim(a(t_i))$, $\forall a$ *that takes place at time* t_i, *and* $\{\mathbf{x}^*(t_i), \mathbf{u}^*(t_i), \mathbf{x}^*(t_{i+1})\}$ *satisfy* $Tran_S(a(t_i))$, $\forall a$ *that takes place at time* t_i, *where* $Lim(a(t_i))$ *is the dynamic limitation of action* a *that takes place at time* t_i, *and* $Tran_S(a(t_i))$ *is the state transition equation of action* a *that takes place at time* t_i*;*

- *The duration of each action satisfies the duration bounds of its action type.* $\forall t_{n_i} \in T'$, $\forall \langle a, d \rangle \in \mathbf{a}^*_{\mathbf{d}}(t_{n_i})$, $d \in [d_l, d_u]$, *where* $[d_l, d_u]$ *are the duration bounds of action* a*;*

- *External constraints are satisfied.* $\forall c_i \in \mathcal{C}$, *the optimal continuous state variables* $\{\mathbf{x}^*(t_1), \mathbf{x}^*(t_2), \ldots, \mathbf{x}^*(t_N)\}$ *satisfy* c_i, *where* $\mathcal{C}$ *is the external constraints;*

- *The objective function* f *is minimized.*

3.3 Problem Statement for $\mathrm{K_{AA}}$

The previous section defined the problem that $\mathrm{K_{DA}}$ solves. In this section, I define the problem that $\mathrm{K_{AA}}$ solves. Recall that $\mathrm{K_{DA}}$ reformulates the durative actions to atomic actions, and then engages $\mathrm{K_{AA}}$. $\mathrm{K_{AA}}$ is the core planner of Kongming. It handles atomic actions.

Definition 22. [Hybrid Atomic Action Planning Problem] *Given* $< \mathbf{s}, \mathbf{u}, \mathcal{I}, \mathcal{G}, \mathcal{AA}, \mathcal{C}, f >$, *the hybrid atomic action planning problem of* K_{AA} *is to return* $\mathcal{P}_{aa}$, *where*

- $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$ *is a set of real-valued and propositional state variables (Def. 2),*

- $\mathbf{u}$ *is a set of real-valued control variables (Def. 3),*

- $\mathcal{I}$ *is the initial conditions (Def. 4),*

- $\mathcal{G}$ *is the final goal conditions (Def. 20),*

- $\mathcal{AA}$ *is a set of hybrid atomic action types (Def. 23),*

- $\mathcal{C}$ *is a set of external constraints (Def. 16),*

- f *is an objective function (Def. 17),*

- $\mathcal{P}_{aa}$ *is a hybrid optimal plan (Def. 27).*

Def. 22 is similar to the hybrid durative action planning problem (Def. 19) that $\mathrm{K_{DA}}$ solves, with two differences. First, the action types in Def. 19 are durative with flexible durations, whereas the action types in Def. 22 are atomic. Second, the output hybrid optimal plan is slightly different.

3.3.1 Input: Hybrid Atomic Action Types

$\mathcal{AA}$ specifies the set of hybrid atomic action types that a system can perform. The actions are hybrid, as they have both continuous and discrete effects. The actions are atomic, as they all have equal and fixed duration.

Definition 23. [Hybrid Atomic Action Type] *A hybrid atomic action type is a 4-tuple, $\langle Cond, Eff, Dyn, d \rangle$, where*

- *$Cond$ is a finite set of* ***conditions*** *of the hybrid atomic action type (Def. 24).*
- *Eff is a finite set of* ***discrete effects*** *of the hybrid atomic action type (Def. 25).*
- *Dyn is the* ***dynamics*** *of the hybrid atomic action type (Def. 13).*
- *$d \in \Re$ is the* ***duration*** *of the hybrid atomic action type (Def. 26).*

Similar to the hybrid durative action types introduced in Section 3.1.4, the atomic action types are parameterized and have the same dynamics as in Def. 13. The differences from the durative action types lie in the conditions, discrete effects and duration. I describe them in detail in the following sections.

In the fire fighting scenario, I described the hybrid durative action types that the air vehicle can perform in Fig. 3-7. The corresponding atomic action types are shown in Fig. 3-8.

Conditions

Recall that conditions are requirements that must be satisfied in order for an action to take place. The definition of a condition of a hybrid atomic action type is similar to that of a hybrid durative action type. The difference is that the condition of a hybrid atomic action type is only specified at the start of the action.

Definition 24. [Condition of a Hybrid Atomic Action Type] *A condition of a hybrid atomic action type is specified by $\mathbf{Ax} \leq b \wedge_i l_i$, where $\mathbf{Ax} \leq b$ is a system of linear inequalities over the continuous state variables $\mathbf{x}$, and l_i is a positive or negative literal for a propositional state variable $p_i \in \mathbf{p}$.*

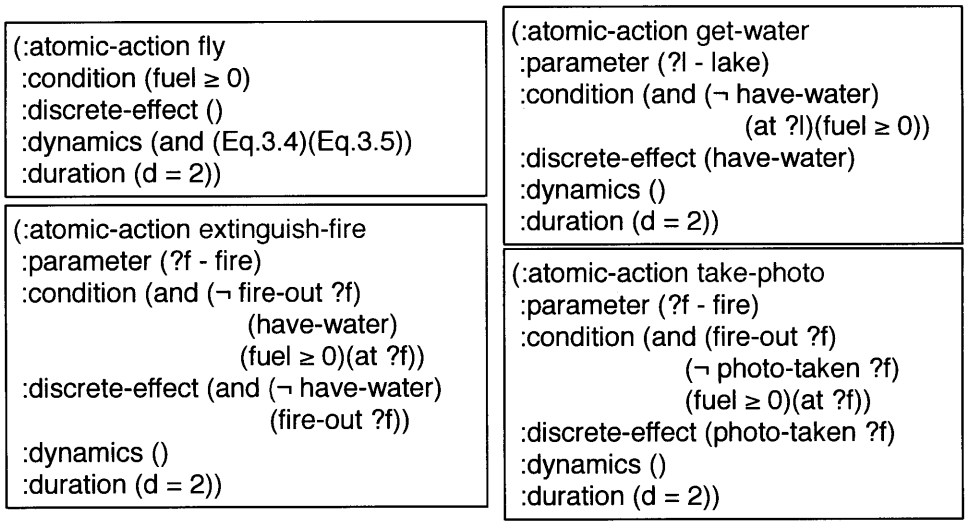


Figure 3-8: In the fire fighting scenario, the air vehicle can perform the following hybrid atomic action types: `fly`, `get-water`, `extinguish-fire`, `take-photo`.

In the fire fighting scenario, as shown in Fig. 3-8, the conditions are similar to those in Fig. 3-7 for durative action types, except that they are not specified over various stages of the actions. For example, the condition of action `get-water` is that the air vehicle has no water, is at the lake and has fuel at the start of the action.

Discrete Effects

Recall that discrete effects capture the changed propositional truth values as a result of the action. The definition of a discrete effect of a hybrid atomic action type is similar to that of a hybrid durative action type. The difference is that the discrete effect of a hybrid atomic action type is only specified at the end of the action.

Definition 25. [Discrete Effect of a Hybrid Atomic Action Type] *A discrete effect of a hybrid atomic action type is specified by* $\wedge_i l_i$*, where* l_i *is a positive or negative literal for a propositional state variable* $p_i \in \mathbf{p}$.

In the fire fighting scenario, as shown in Fig. 3-8, the discrete effects are similar to those in Fig. 3-7 for durative action types, except that they are not specified over various stages of the actions. For example, the discrete effect of action `get-water` is that the air vehicle has water at the end of the action.

Duration

The atomic action types all share the same fixed duration.

Definition 26. [Duration of a Hybrid Atomic Action Type] *The duration of a hybrid atomic action type is a fixed real-valued number, $d \in \Re$.*

In the fire fighting scenario, as shown in Fig. 3-8, all action types have duration $d = 2$.

3.3.2 Output: Optimal Hybrid Plan

I define the optimal hybrid plan output of K_{AA} in this section.

Definition 27. [Optimal Hybrid Plan of K_{AA}] *An optimal hybrid plan of K_{AA} is a triple, $\mathcal{P}_{AA} =< \mathbf{A}^*, \mathbf{S}^*, \mathbf{U}^* >$, where*

- $\mathbf{A}^* = \{\mathbf{a}^*(t_1), \mathbf{a}^*(t_2), \ldots, \mathbf{a}^*(t_{N-1})\}$ *is an optimal sequence of actions instantiated from the hybrid atomic action types $\mathcal{AA}$, $\mathbf{a}^*(t_i)$ is the set of actions to be performed at time t_i;*
- $\mathbf{S}^* = \{\mathbf{s}^*(t_1), \mathbf{s}^*(t_2), \ldots, \mathbf{s}^*(t_N)\}$ *is an optimal sequence of assignment to the continuous and discrete state variables $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$, $\mathbf{s}^*(t_i)$ is the assignment to $\mathbf{s}$ at time t_i;*
- $\mathbf{U}^* = \{\mathbf{u}^*(t_1), \mathbf{u}^*(t_2), \ldots, \mathbf{u}^*(t_{N-1})\}$ *is an optimal sequence of assignment to the control variables $\mathbf{u}$, $\mathbf{u}^*(t_i)$ is the assignment to $\mathbf{u}$ at time t_i;*

such that all of the following are true:

- $\forall i = 1, \ldots, N-1, t_{i+1} - t_i = \Delta t$, *where Δt is a discretization constant.*
- $< \mathbf{A}^*, \mathbf{S}^*, \mathbf{U}^* >$ *is a valid hybrid plan (Def. 36).*
- *The objective function f is minimized.*

This chapter formally defined the problem statement for the three planners in Kongming: K_{QSP}, K_{DA} and K_{AA}. Chapter 4 will introduce the plan representation of K_{AA}, and Chapter 5 will introduce the planning algorithm of K_{AA}.

Chapter 4

Plan Representation for K_{AA}

Contents

As defined in Chapter 3, Kongming plans for a qualitative state plan (QSP) with temporally flexible hybrid actions. Recall that Kongming is divided into three planners, as shown in Fig. 4-1. K_{QSP} is a planner that plans for QSPs, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. K_{DA} is a planner that handles durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. K_{AA} is a planner that handles atomic actions and a final goal state. I describe the plan representation for K_{AA} in this chapter.

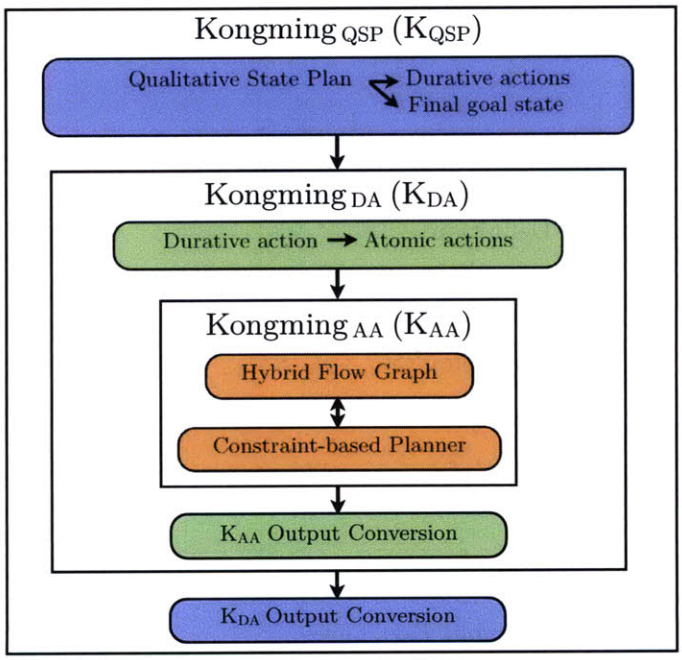


Figure 4-1: Overview of Kongming's approach.

K_{AA} consists of the following two parts: 1) a compact representation of the space of possible hybrid plans (plan space), which can be employed by a wide range of planning algorithms, from constraint-based planning [WW99, SD05, KS92, KS99, DK01a] to heuristic search [CCFL09, HN01, MBB+09, DK01b, BG99, BG01, McD96]; and 2) a constraint-based planning algorithm that operates on the hybrid plan representation. I introduce the plan representation in this chapter, and the constraint-based planning algorithm in the next chapter.

Kongming's plan representation, called the Hybrid Flow Graph, builds upon the Planning Graph from Graphplan [BF97] and Flow Tubes [Hof06]. Flow tubes represent all valid trajectories of continuous actions, starting from an initial region. A flow tube is followed by a flow tube or a discrete action, if the first flow tube's end region has a non-empty intersection with the follower's continuous condition. Similar to a Planning Graph, a Hybrid Flow Graph is a directed, leveled graph that alternates between *fact levels* and *action levels*. Different from a Planning Graph, fact levels in a Hybrid Flow Graph contains not only propositional facts but also continuous regions, and action levels contain hybrid actions. A Hybrid Flow Graph represents all valid plans while pruning some invalid plans through mutual exclusion.

This chapter is organized as follows. First, I introduce the flow tube representation of hybrid actions. Second, I introduce Hybrid Flow Graphs to represent all valid plans. Finally, I describe the algorithm for constructing a Hybrid Flow Graph.

4.1 Flow Tube Representation of Actions

Recall the discussion in Section 2.4. When planning with purely discrete actions, as in Graphplan [BF97] and other discrete planners, the decision variables are discrete, and hence the planner only needs to reason about a finite number of trajectories in plan space. However, when continuous actions and continuous states are included in planning, the planner needs a compact way of representing and reasoning about an infinite number of trajectories and states. Kongming's representation of each set of trajectories is a *flow tube* in the spirit of [Hof06].

In this section, I address the problem of how to compactly represent the infinite number of trajectories of each continuous action. More specifically, the problem I am solving is, given an initial region in state space, defined by a conjunction of linear (in)equalities over the continuous state variables, $\mathbf{x}$, and given an action model, including continuous conditions, a state equation of the system dynamics, actuation limits, and a duration, generate an abstract description of all possible state trajectories of this action, starting from the initial region. I justify this problem statement as

follows.

- Each action is assumed to start from an initial region, which is more general than an initial point. Due to execution uncertainty, the initial state of an action, when it is executed, is uncertain. This uncertainty is partly addressed by keeping a region for the initial state.

- In this section, the initial region is considered as given, and the focus is on the core concepts of generating flow tubes. Section 4.2.3 addresses the computation of initial regions, in the context of the Hybrid Flow Graph.

- Recall the definition of hybrid atomic action types in Chapter 3. A hybrid atomic action type consists of four components: conditions, discrete effects, dynamics, and duration. The continuous condition in the action model constrains the initial region from where the action starts, which will be discussed in Section 4.2.3. The discrete conditions and effects in the action model do not affect the continuous state trajectory of the action, and hence, are not considered. The dynamics are derived from the physical properties of the autonomous system performing the action. More specifically, the state equation governs the evolution of the continuous state variables from one time point to the next. The actuators of the autonomous system, for example, electric motors, have limited power, and hence, there are actuation limits.

- Recall from the problem statement for K_{AA} in Section 3.3, there are external constraints, $\mathcal{C}$, which impose external requirements on the state of the system. For example, $\mathcal{C}$ could include constraints requiring that the position of a vehicle be outside an obstacle, which are expressed as a disjunction of linear (in)equalities over the state variables. However, in this section, I simplify the problem of representing the possible state trajectories of an action by ignoring these constraints, so that I can focus on the core concepts of flow tubes. The constraints will be discussed in Section 4.2.2.

- I am interested in a description of all feasible state trajectories that an action

could evolve through, in order to have a complete plan representation. K_{AA} engages its constraint-based planner on the complete plan representation, to search for an optimal plan, which will be described in Chapter 5.

K_{AA} uses a flow tube as the abstract description of all possible state trajectories of each *continuous action.* I call a hybrid action that has specified dynamics a *continuous action.* Intuitively, a continuous action, alters or maintains the continuous state of the autonomous system over time. I start the explanation with the simplest case. A continuous action starting from a specific initial state, with a specific control input and a specific duration, maps to a state trajectory of the action, connecting the initial state and the end state of the system. For example, as shown in Fig. 4-2, the x axis is the continuous state space of an AUV, and the t axis is time. The `glide` action with control value v between t_1 and t_2 corresponds to the line connecting point (t_1, x_1) and point (t_2, x_2).

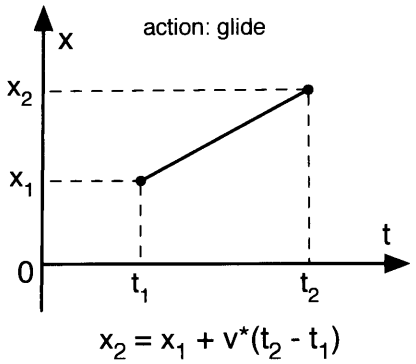


Figure 4-2: `Glide` action with control input value v between t_1 and t_2 corresponds to the line connecting point (t_1, x_1) and point (t_2, x_2).

Autonomous systems in general can have a range of control input values. For example, an AUV's x-velocity may be directly controllable and have range $[-2m/s, 2m/s]$. Thus an action with a range of control values for a specific duration maps to a range of state trajectories. I use a flow tube to represent the range of trajectories. For example, as seen in Fig. 4-3, line AB represents the trajectory for control value v_{max} and line AC represents the trajectory for control value v_{min}. Triangle ABC is the flow tube used to represent the range of trajectories. Line BC is called the cross section at time

t_2 of action `glide` that starts from initial state x_1 at t_1. A flow tube is a collection of all the cross sections between two time points.

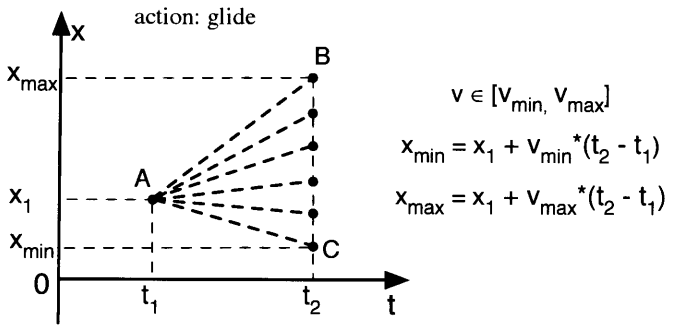


Figure 4-3: Line AB corresponds to the trajectory for control value v_{max} and line AC corresponds to the trajectory for control value v_{min}. Triangle ABC is the flow tube used to represent the range of trajectories.

4.1.1 Flow Tube Definition

I define a *flow tube* through the concept of *cross sections*. Given an initial region, an action model and a duration, a cross section *cs* is the set of all reachable states over the duration.

Definition 28. [Cross Section] *Given a time point t_i and a non-negative duration d, given the state equation of an action type V, $\mathbf{x}(t_i + d) = \mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i)$, given time invariant bounds on the control variables $\mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$, and given an initial region $R_I = \{\mathbf{x} \in \Re^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$, where $\mathbf{x}(t_i) \in R_I$ and each $g_j(\mathbf{x})$ is a linear function over $\mathbf{x}$, the* **cross section** *of duration d of a V type action that starts from R_I, $cs(R_I, d)$, is the range of function*

$$\mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i), \text{ where } \mathbf{x}(t_i) \in R_I \text{ and } \mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]. \tag{4.1}$$

For the special case where $d = 0$, $cs(R_I, d) = R_I$.

The properties of a cross section *cs* largely depend on the properties of its initial region R_I. Hence, I first describe the properties of R_I. Because the initial region R_I is a polytope, defined by a conjunction of linear inequalities $\wedge_j g_j(\mathbf{x}) \leq 0$, it has the

nice properties associated with a polytope. I describe the properties here. For formal proofs, please refer to [BT97]. If R_I is a non-empty polytope, then

- R_I can always be represented a polytope with at least one vertex (extreme point).
- Let $\mathbf{x}^1, \ldots, \mathbf{x}^p$ be the vertices of R_I, and $\mathbf{w}^1, \ldots, \mathbf{w}^q$ be a complete set of extreme rays of R_I. Then

$$R_I = \left\{ \sum_{k=1}^{p} \lambda_k \mathbf{x}^k + \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\}. \tag{4.2}$$

Eq. 4.2 states that every point in R_I can be represented by a convex combination of the vertices of R_I and a linear combination of the extreme rays of R_I. Because it is trivial to compute the range of function $\mathbf{Ax}(t_i) + \mathbf{Bu}(t_i)$, where $\mathbf{x}(t_i)$ is a point and $\mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$, this property provides an exact way of computing cross section $cs(R_I, d)$. I provide the equations for computing the cross section in the following lemma, and prove its correctness.

Lemma 1. *Let t_i be a time point and d be a non-negative duration. Let the state equation of an action type V be $\mathbf{x}(t_i + d) = \mathbf{Ax}(t_i) + \mathbf{Bu}(t_i)$. Let $R_I = \{\mathbf{x} \in \Re^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$ be an initial region, such that $\mathbf{x}(t_i) \in R_I$. Let R_I be a non-empty polytope with at least one vertex. Let $\mathbf{x}^1, \ldots, \mathbf{x}^p$ be the vertices, and $\mathbf{w}^1, \ldots, \mathbf{w}^q$ be a complete set of extreme rays of R_I. Let $cs(\mathbf{x}^k, d)$ be the cross section of duration d of a V type action starting from an initial state $\mathbf{x}^k$. Let $cs(R_I, d)$ be the cross section of duration d of a V type action starting from R_I. Then all of the following hold:*

$$cs(R_I, d) = \left\{ \sum_{k=1}^{p} \lambda_k \, cs(\mathbf{x}^k, d) + \mathbf{A} \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \tag{4.3}$$

$$\left\{\mathbf{w}^1, \ldots, \mathbf{w}^q\right\} = \emptyset \Rightarrow cs(R_I, d) = \left\{ \sum_{k=1}^{p} \lambda_k \, cs(\mathbf{x}^k, d) \mid \lambda_k \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \tag{4.4}$$

Proof. (4.3): From Eq. 4.2, we know that any point in R_I can be represented as $\sum_{k=1}^{p} \lambda_k \mathbf{x}^k + \sum_{l=1}^{q} \theta_l \mathbf{w}^l$, where $\lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1$. From Definition 28, we

know that $cs(R_I, d)$ is the range of function $\mathbf{Ax}(t_i) + \mathbf{Bu}(t_i)$, where $\mathbf{x}(t_i) \in R_I, \mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$. Hence,

$$
\begin{aligned}
cs(R_I, d) &= \left\{ \mathbf{A}(\sum_{k=1}^{p} \lambda_k \mathbf{x}^k + \sum_{l=1}^{q} \theta_l \mathbf{w}^l) + \mathbf{Bu}(t_i) \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \\
&= \left\{ \sum_{k=1}^{p} \lambda_k \mathbf{Ax}^k + \mathbf{Bu}(t_i) + \mathbf{A} \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \\
&= \left\{ \sum_{k=1}^{p} \lambda_k \mathbf{Ax}^k + \mathbf{Bu}(t_i) \sum_{k=1}^{p} \lambda_k + \mathbf{A} \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \\
&= \left\{ \sum_{k=1}^{p} \lambda_k (\mathbf{Ax}^k + \mathbf{Bu}(t_i)) + \mathbf{A} \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\} \\
&= \left\{ \sum_{k=1}^{p} \lambda_k cs(\mathbf{x}^k, d) + \mathbf{A} \sum_{l=1}^{q} \theta_l \mathbf{w}^l \mid \lambda_k \geq 0, \theta_l \geq 0, \sum_{k=1}^{p} \lambda_k = 1 \right\}
\end{aligned}
$$

(4.4): Eq. 4.4 naturally follows Eq. 4.3. □

A *flow tube* is a set of all the cross sections between two time points. This set of cross sections is infinite and is described analytically through a function of time.

Definition 29. [Flow Tube] *Given a time point t_i and a non-negative duration d, the state equation of an action type $\mathbf{x}(t_i + d) = \mathbf{Ax}(t_i) + \mathbf{Bu}(t_i)$, time invariant bounds on the control variables $\mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$, and an initial region $R_I = \{\mathbf{x} \in \Re^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$, where $\mathbf{x}(t_i) \in R_I$ and each $g_j(\mathbf{x})$ is a linear function over $\mathbf{x}$, a* **flow tube** *of the action that starts from initial region R_I is a function $ft(R_I, t)$ defined in the interval of $[0, d]$ such that*

$$ft(R_I, t) = cs(R_I, t), t \in [0, d]. \tag{4.5}$$

$ft(R_I, 0) = R_I$ is called the **initial region** *of the flow tube, and $ft(R_I, d)$ is called the* **end region** *of the flow tube.*

Lemma 1 provides the equations for computing the cross section. It says, when the initial region is bounded and has n corners, the cross section of the initial region

over a duration, is the convex hull of all n cross sections, each of which is computed from a corner of the initial region.

Fig. 4-4 shows an example of how to compute a flow tube of an action in 1D for a second-order acceleration limited dynamic system. The state space is the position-velocity space, $\mathbf{x} =< x, \dot{x} >$. The control input is acceleration, $\mathbf{u} =< a_x >$. Starting time point is t_i, and duration $d = 1$. The state equation is

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix} (t_i + d) = \begin{bmatrix} 1 & d \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ \dot{x} \end{bmatrix} (t_i) + \begin{bmatrix} \frac{d^2}{2} \\ d \end{bmatrix} \times \begin{bmatrix} a_x \end{bmatrix} (t_i) \tag{4.6}$$

The actuation limit is bounds on acceleration, $a_x \in [-1, 2]$. The initial region R_I is shown in Fig. 4-4(a) in the position-velocity space, with its 4 corners marked A, B, C and D. The cross section for each corner is computed, based on Eq. 4.6 and Def. 28. The cross section for corner A is region $\{x \in [2.5, 4], \dot{x} \in [0, 3]\}$, for B is region $\{x \in [3.5, 5], \dot{x} \in [0, 3]\}$, for C is region $\{x \in [5.5, 7], \dot{x} \in [2, 5]\}$, and for D is region $\{x \in [4.5, 6], \dot{x} \in [2, 5]\}$, as shown in Fig. 4-4(b). Finally, the convex hull of the 4 cross sections is computed as the cross section of initial region R_I. The convex hull is shown in the figure as the union of the 4 cross sections and additional regions specified by the slanted dashed lines, as shown in Fig. 4-4(b). Fig. 4-4(c) shows the resulting flow tube.

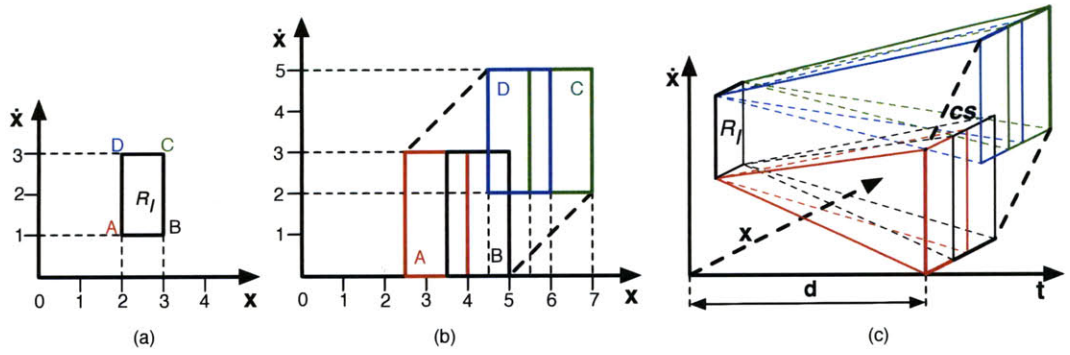


Figure 4-4: An example of constructing a flow tube of an action in 1D for a second-order acceleration limited dynamical system. (a) shows the initial region R_I. (b) shows the cross section of R_I. (c) shows the flow tube of duration d.

4.1.2 Properties

In this section, I discuss the properties of cross sections and flow tubes. A cross section contains and only contains all the reachable points. A flow tube contains all the valid trajectories, but also contain some invalid trajectories.

Based on the definition of cross sections, I describe two important properties.

Lemma 2. *Let t_i be a time point and d be a non-negative duration. Let the state equation of an action type V be $\mathbf{x}(t_i + d) = \mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i)$. Let time invariant bounds on the control variables be $\mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$. Let $cs(R_I, d)$ be the cross section of duration d of a V type action starting from initial region R_I. Then the following hold:*

(a) *$cs(R_I, d)$ contains all points that are reachable at $t_i + d$ by the action from R_I.*

(b) *$cs(R_I, d)$ only contains points that are reachable at $t_i + d$ by the action from R_I.*

Proof. (a). Suppose there exists outside $cs(R_I, d)$, a point $\mathbf{x}^* \in \mathcal{R}^n$ that is reachable at $t_i + d$ from R_I. $\mathbf{x}^*$ is a reachable point, so $\mathbf{x}^*$ satisfies $\mathbf{x}^* = \mathbf{A}\bar{\mathbf{x}} + \mathbf{B}\bar{\mathbf{u}}$, where $\bar{\mathbf{x}} \in R_I$ and $\bar{\mathbf{u}} \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$. Because $cs(R_I, d)$ is the range of function $\mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i)$, where $\mathbf{x}(t_i) \in R_I, \mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$, we know $\mathbf{A}\bar{\mathbf{x}} + \mathbf{B}\bar{\mathbf{u}} \in cs(R_I, d)$. Hence $\mathbf{x}^* \in cs(R_I, d)$, which contradicts the assumption. Therefore, there does not exist such a point $\mathbf{x}^*$.
(b). Suppose there exists a point $\mathbf{x}^* \in \mathcal{R}^n$ in $cs(R_I, d)$ that is not reachable at $t_i + d$ from R_I. Because $\mathbf{x}^* \in cs(R_I, d)$, we know

$$\mathbf{x}^* \in \{ \text{ range of function } \mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i) \mid \mathbf{x}(t_i) \in R_I, \mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}] \} .$$

Hence, there must exist a $\bar{\mathbf{x}} \in R_I$ and $\bar{\mathbf{u}} \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$, such that $\mathbf{x}^* = \mathbf{A}\bar{\mathbf{x}} + \mathbf{B}\bar{\mathbf{u}}$. Therefore, $\mathbf{x}^*$ is reachable, which contradicts the assumption. □

Based on the definition of flow tubes, I describe two important properties.

Lemma 3. *Let t_i be a time point and d be a non-negative duration. Let $cs(R_I, t), t \in [0, d]$ be cross sections of duration from 0 to d, of a type V action that starts at t_i*

from initial region R_I. Let $ft(R_I, t)$ be the flow tube of the type V action. Then the following hold:

(**a**) *$ft(R_I, t)$ contains all the valid trajectories.*

(**b**) *$ft(R_I, t)$ also contains invalid trajectories.*

Proof. (a). Suppose there exists a valid trajectory $traj(t_i, t_i + d)$ of the action between two time points t_i and $t_i + d$, and it goes outside $ft(R_I, t)$. A trajectory is a function of time. $traj(t_i, t_i + d) = y(t), t \in [t_i, t_i + d]$. The image of each time t through the function is an assignment to the state variables $\mathbf{x}$. A flow tube is a function of time (duration), $ft(R_I, t)$. The image of each time t through the function is a cross section, $ft(R_I, t) = cs(R_I, t), t \in [0, d]$. Since $traj(t_i, t_i + d)$ goes outside $ft(R_I, t)$, there must exist at least a time point t^*, such that $y(t^*)$ is not contained in $cs(R_I, t^* - t_i)$. Because $traj(t_i, t_i + d)$ is a valid trajectory, $y(t^*)$ is a reachable point. This contradicts Lemma 2(a), hence the assumption is false.

(b). I give two examples of the invalid trajectories. As shown in Fig. 4-5, inside flow tube $ft(R_I, t)$, there are two invalid trajectories. The straight dashed line satisfies the state equation, but its control variable value is outside the actuation limit. The curvy solid line satisfies neither the state equation nor the actuation limit. □

4.1.3 Flow Tube Approximation

Computing a flow tube is equivalent to computing its cross sections, Def. 29. Lemma 1 provides an exact way of computing a cross section. However, it requires enumerating every vertex and extreme ray of the initial region R_I, and taking the convex hull. As shown in [KV07, ABS97], even in low dimensional systems the number of vertices can grow very large with the number of *steps*. The number of *steps* in the context of this thesis means the number of consecutive flow tubes in the Hybrid Flow Graph. When there are a large number of vertices or extreme rays, computing the cross section becomes expensive. In this thesis, the cross sections are approximated with external

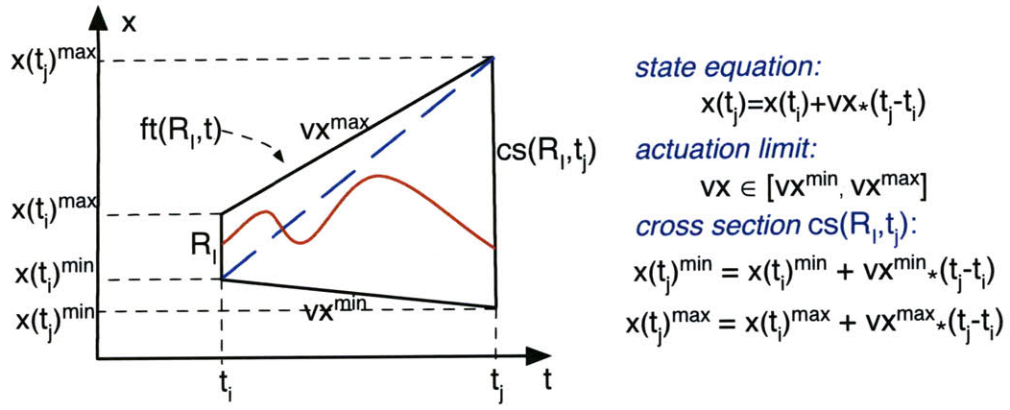


Figure 4-5: Examples of invalid trajectories in a flow tube. There is one state variable, x. There is one control variable, vx. The state equation and actuation limit of the action are listed on the right. Inside flow tube $ft(R_I, t)$, two invalid trajectories are shown. The straight dashed line satisfies the state equation, but its control variable value is outside the actuation limit. The curvy solid line satisfies neither the state equation nor the actuation limit.

orthotopes[1].

I define the approximate cross section as follows.

Definition 30. [Approximate Cross Section] *Given a time point t_i and a non-negative duration d. Given the state equation of an action type V, $\mathbf{x}(t_i + d) = \mathbf{A}\mathbf{x}(t_i) + \mathbf{B}\mathbf{u}(t_i)$. Given time invariant bounds on the control variables $\mathbf{u}(t_i) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}]$. Given an initial region $R_I = \{\mathbf{x} \in \Re^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$, where $\mathbf{x}(t_i) \in R_I$ and each $g_j(\mathbf{x})$ is a linear function over $\mathbf{x}$. Let $\mathbf{x}_{lb}(t_i) \in \mathcal{R}^n$ be the lower bound of $\mathbf{x}(t_i)$, and $\mathbf{x}_{ub}(t_i) \in \mathcal{R}^n$ be the upper bound of $\mathbf{x}(t_i)$. The* **approximate cross section**, *$\widetilde{cs}(R_I, d)$, is*

$$\widetilde{cs}(R_I, d) = \{\mathbf{x} \in \mathcal{R}^n \mid \mathbf{x} \in [\mathbf{x}_{lb}, \mathbf{x}_{ub}]\}, \text{ where} \tag{4.7}$$

$\mathbf{x}_{lb} = \mathbf{A}\mathbf{x}_{lb}(t_i) + \mathbf{B}\mathbf{u}_{lb}$, *and* $\mathbf{x}_{ub} = \mathbf{A}\mathbf{x}_{ub}(t_i) + \mathbf{B}\mathbf{u}_{ub}$.

The approximate cross section for the example in Fig. 4-4 is computed as follows. The state equation is in Eq. 4.6. The initial region R_I is shown in Fig. 4-4(a). As mentioned previously, the state space is the position-velocity space, $\mathbf{x} =< x, \dot{x} >$, the

[1] An orthotope is an n-dimensional generalization of a rectangular region.

control variable is acceleration, $\mathbf{u} =< a_x > \in [-1, 2]$, and duration $d = 1$. According to Def. 30, we know $\mathbf{x}_{lb}(t_i) =< x_{lb}(t_i), \dot{x}_{lb}(t_i) >=< 2, 1 >$, $\mathbf{x}_{ub}(t_i) =< x_{ub}(t_i), \dot{x}_{ub}(t_i) >=< 3, 3 >$, and we can compute $\mathbf{x}_{lb} = \mathbf{A}\mathbf{x}_{lb}(t_i) + \mathbf{B}\mathbf{u}_{lb} =< 2.5, 0 >$, and $\mathbf{x}_{ub} = \mathbf{A}\mathbf{x}_{ub}(t_i) + \mathbf{B}\mathbf{u}_{ub} =< 7, 5 >$. The resulting approximate cross section of the cross section in Fig. 4-4(b) is shown in Fig. 4-6.

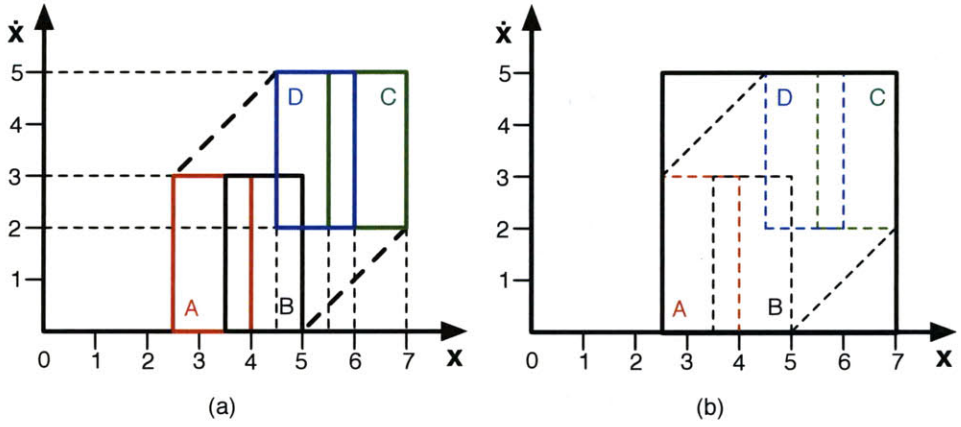


Figure 4-6: (a) shows the exact cross section. (b) shows the orthotope external approximation of the cross section.

Intuitively, an approximate cross section is an orthotope that contains its corresponding exact cross section, $\widetilde{cs}(R_I, d) \supseteq cs(R_I, d)$. Based on Lemma 2, the approximate cross section contains all reachable points of the action, and may also contain points that are not reachable.

4.1.4 Related Work

Hofmann [Hof06] uses flow tubes to represent bundles of state trajectories that take into account dynamic limitations due to under-actuation, while satisfying plan requirements for the foot placement of walking bipeds. By defining the valid operating regions for the state variables and control parameters in the abstracted model of a biped, the flow tubes prune infeasible trajectories and ensure plan execution.

Kongming is similar to [Hof06] in that Kongming also uses flow tubes to represent bundles of state trajectories that satisfy dynamic limitations and plan requirements. Flow tubes in Kongming are different from those in [Hof06] in the following aspects.

- Flow tubes in Kongming contain all valid trajectories; they are complete. Flow tubes in [Hof06] exclude some of the valid trajectories; they are incomplete.

- Flow tubes in Kongming contain invalid trajectories, while flow tubes in [Hof06] only contain valid trajectories.

- Two flow tubes in Kongming are connected, if the end region of the first flow tube intersects with the continuous condition of the second flow tube, as described in Section 4.2. Two flow tubes in [Hof06] are connected, if the end region of the first flow tube is a subset of the initial region of the second flow tube.

- In Kongming, a flow tube can also be connected to a discrete action, if the end region of the flow tube intersects with the continuous condition of the discrete action. There are no discrete actions in [Hof06].

Similar representations are used in the robotics community, under the name funnels [Ted09], and in the hybrid systems community, under reachability analysis [KV07, KGBM04, Gir05, SK03, Kos01]. Reachability analysis for general polytopes is implemented in the Multi-Parametric Toolbox (MPT) for Matlab [KGBM04], which is used by [Hof06]. It computes the geometric sum of polytopes at every time step, by finding the vertices and calculating the convex hull. As pointed out by [KV07, ABS97], this is very computationally intensive. [Gir05] uses zonotopes for external approximation of the reach sets. [SK03] uses oriented rectangular polytopes for external approximation. It is similar to the approximation method K_{AA} uses in that both methods use rectangular polytopes for external approximation. However, [SK03] orients the rectangular polytopes to best approximate the reach sets. It gives a better approximation, at the cost of more computation effort. [Kos01] approximates the reach sets with parallelotopes. [KV07] approximate with external and internal ellipsoids.

4.2 Hybrid Flow Graph

Thus far I have described how to represent hybrid actions as flow tubes, and the computation and approximation of flow tubes. In this section, I introduce how to incorporate the flow tubes into a graph structure similar to a Planning Graph [BF97]. The resulting graph is called a Hybrid Flow Graph.

4.2.1 Planning Graph

Because the Hybrid Flow Graph builds upon a Planning Graph [BF97] structure, I first review a subset of a Planning Graph that is relevant to the Hybrid Flow Graph. Additional review can be found in Section 2.5.

The Planning Graph introduced in Graphplan [BF97], is a compact structure for representing the plan space in the STRIPS-like planning domain. It has been employed by a wide range of modern planners, including LPGP [LF02], STAN [LF99], GP-CSP [DK01a], to name a few. The Graphplan planner constructs, analyses the structure, and extracts a valid plan from it. A Planning Graph encodes the planning problem in such a way that many useful constraints inherent in the problem become explicitly available to reduce the amount of search needed. Planning Graphs can be constructed quickly: they have polynomial size and can be built in polynomial time.

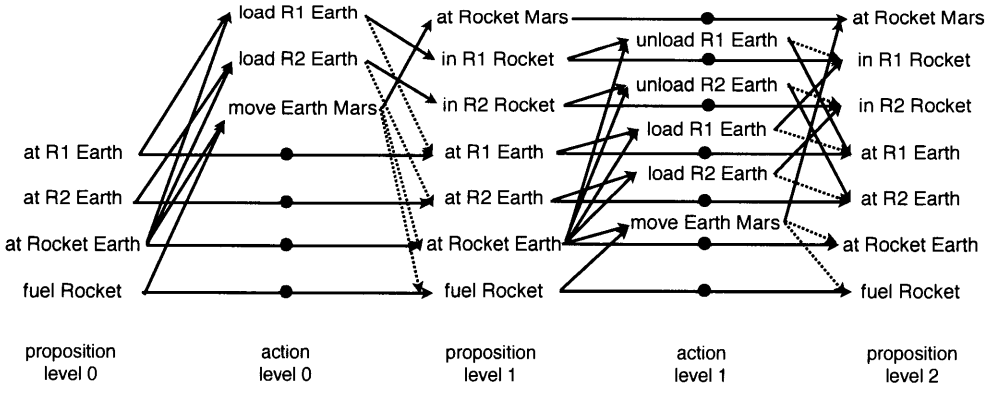


Figure 4-7: A Planning Graph of 3 proposition levels and 2 action levels. Round black dots represent no-op actions. Solid lines connect conditions with actions and actions with add-effects. Dotted lines connect actions with delete-effects.

As shown in Fig. 4-7, a Planning Graph is a directed, leveled graph, which alternates between proposition levels, containing propositions, and action levels, containing action instantiations. Proposition level i contains all propositions that can be true at time i, and action level i contains all possible actions to take place at time i. There are special actions, called *no-op* actions. They preserve propositions from one proposition level to the next.

Propositions in proposition level i are connected to actions in action level i if the propositions are the conditions of the actions. For example, in Fig. 4-7, `at` R_1 `Earth` and `at Rocket Earth` in proposition level 0 are connected to `load` R_1 `Earth` in action level 0, because `at` R_1 `Earth` and `at Rocket Earth` are the conditions of action `load` R_1 `Earth`.

Actions in action level i are connected to propositions in proposition level $i+1$ if the propositions are the effects of the actions. For example, in Fig. 4-7, `load` R_1 `Earth` in action level 0 is connected to `in` R_1 `Rocket` and `at` R_1 `Earth` in proposition level 1, because `in` R_1 `Rocket` is the add-effect of `load` R_1 `Earth` and `at` R_1 `Earth` is the delete-effect of `load` R_1 `Earth`. Add-effects are connected using solid lines, and delete-effects are connected using dotted lines. Because no-op actions exist, every proposition that appears in proposition level i also appears in proposition level $i+1$.

Similar to a Planning Graph, a Hybrid Flow Graph is a directed, leveled graph that alternates between *fact levels* and *action levels*. Fact level i contains all facts that can be true at time i, and action level i contains all possible actions that can take place at time i. Different from a Planning Graph, a fact level contains both propositional facts and continuous regions, and an action level in a Hybrid Flow Graph contains hybrid actions.

In a Hybrid Flow Graph, the no-op actions do not preserve the continuous regions in a fact level. This is because a no-op action for a continuous region means that the state variables do not change their values unless an action explicitly changes them. For a hybrid autonomous system, this means that the system is assumed to be able to *stop*. This is a reasonable assumption for certain systems, such as vehicles that move very slowly. However, it is not a reasonable assumption for systems like bi-peds, many

AUVs, or air vehicles. Therefore, no-op actions in a Hybrid Flow Graph are only for preserving literals. If a system can *stop*, it can have a hybrid action type defined in the input, called `maintain`. The function of `maintain` would be maintaining the values of the state variables. Flow tubes of this action type would have equivalent initial regions and end regions.

In the following sections, I introduce the fact and action levels of a Hybrid Flow Graph, and then describe the construction of the Hybrid Flow Graph.

4.2.2 Hybrid Flow Graph: Fact Levels

Recall that there are two types of levels in a Hybrid Flow Graph: fact levels and action levels. In this section, I introduce fact levels. Different from in a Planning Graph, a fact level in a Hybrid Flow Graph contains not only propositional facts that are positive or negative literals, but also continuous regions that are specified by conjunctions of linear inequalities over state variables.

There are two sources for the positive or negative literals.

- The discrete initial conditions. As defined in Section 3.3, they are part of the initial conditions, specified by a set of literals.

- Discrete effects of actions. As defined in Section 3.3, the discrete effects of a hybrid action type are a set of literals. No-op actions preserve literals from the previous fact level to the next. Hence, the effects of no-op actions are also literals.

In Section 4.1 when flow tubes are introduced, I ignore the existence of external constraints $\mathcal{C}$, in order to focus on the core concepts of flow tubes. I now take these constraints into consideration.

As defined in Section 3.3, the constraints are described by a conjunction of clauses, each of which is either a disjunction of linear inequalities over the state variables, or a unit clause, which is a linear inequality over the state variables. For example, a constraint can require that the position of a vehicle be outside an obstacle, which is

expressed as a disjunction of linear inequalities over the state variables. A constraint can also require that the position of a vehicle be within a region, which is expressed as a conjunction of linear inequalities over the state variables.

As described in Section 4.1, a cross section is all reachable points of an action without considering the constraints. The end region of flow tube is a cross section, and the intersection of the end region and the constraints represents the set of all the reachable points of an action that satisfy the constraints. Because there are non-unit clauses in the constraints, the region that the constraints represent is not convex. Hence, the intersection of a cross section and the constraint region is not necessarily convex. This will break the convexity assumption for initial regions R_I in computing cross sections and flow tubes. Therefore, I ignore the non-unit clauses and keep the unit clauses in the constraints when taking the intersection. However, because the intersection region ignores the non-unit clauses in the constraints, the region can contain points in state space that violate some of the constraints.

Definition 31. [Resolved End Region] *Let R_g be the end region of flow tube ft, and let $\mathcal{C}$ be the external constraints defined in the input. The resolved end region R_{rg} of ft is, $R_{rg} = R_g \bigcap \mathcal{C}'$, where $\mathcal{C}'$ is $\mathcal{C}$ without non-unit clauses.*

Because both R_g and $\mathcal{C}'$ are convex regions, specified by a conjunction of linear inequalities over the state variables, their intersection, R_{rg}, is also a convex region, specified by a conjunction of linear inequalities over the state variables.

There are two sources for a continuous region in a fact level.

- The continuous initial condition. As defined in Section 3.3, it is a continuous region, specified by a conjunction of linear inequalities over the state variables.
- The resolved end region of a flow tube. It is a continuous region, specified by a conjunction of linear inequalities over the state variables.

Note that a fact level contains a limited collection of literals, as they are pre-defined in the input. This is also the case in a Planning Graph. However, the resolved end regions in a fact level are not pre-defined. There is an unlimited collection of

continuous regions that can be included in a fact level. This affects the termination property of the Hybrid Flow Graph, as described later in Section 4.3.5.

Definition 32. [Fact Level] *Given an action level at $i \geq 0$, $AL(i)$, a fact level at $i+1$, $FL(i+1)$, is $< P, R >$, where $P = \{p_1, p_2, \ldots\}$ is a set of literals that either are discrete effects of actions in $AL(i)$ or are literals preserved by no-op actions in $AL(i)$. $R = \{r_1, r_2, \ldots\}$ is a set of continuous regions, each specified by a conjunction of linear inequalities over state variables, $\wedge_j g_j(\mathbf{x}) \leq 0$. Each r is a resolved end region of a flow tube in $AL(i)$. When $i = 0$, $AL(i)$ does not exist, and $FL(i+1) = FL(1)$ is the first level in the graph. In $FL(1)$, P is the set of literals in the initial conditions and R is the continuous region in the initial conditions.*

Fig. 4-8(a) shows an example of a fact level. It contains literals and continuous regions.

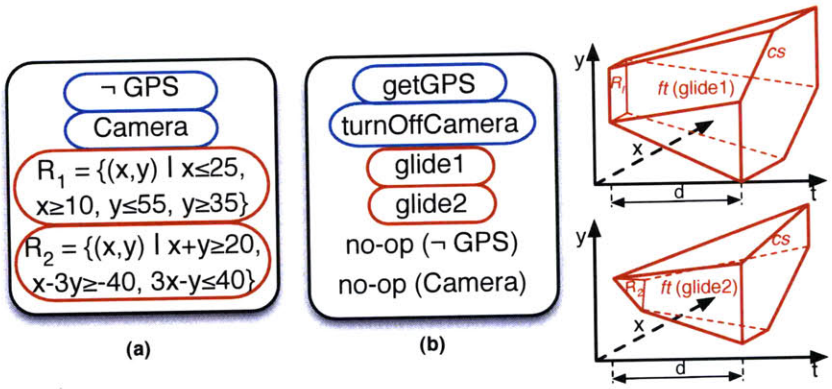


Figure 4-8: (a) shows an example of a fact level. It contains literals, circled in blue, and continuous regions, circled in red. (b) shows an example of an action level. It contains instantiations of hybrid action types. The ones with dynamics are represented by flow tubes. It also contains no-op actions.

4.2.3 Hybrid Flow Graph: Action Levels

Recall that there are two types of levels in a Hybrid Flow Graph: fact levels and action levels. I have described fact levels thus far; next I introduce action levels. An action level consists of instantiations of hybrid action types defined in the input (Section

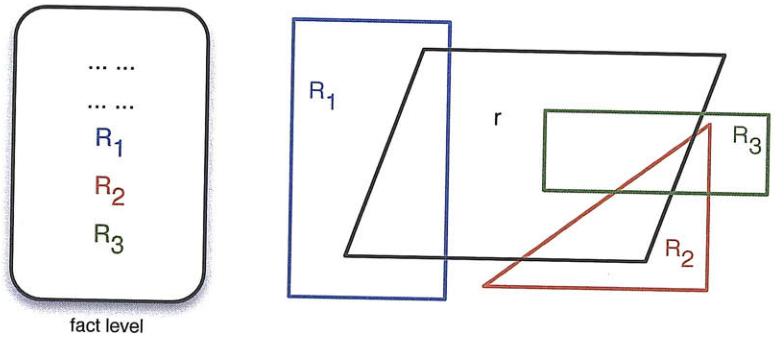


Figure 4-9: Suppose R_1, R_2 and R_3 are in a fact level. As R_1, R_2 and R_3 all have nonempty intersection with a region r, R_1, R_2 and R_3 are all resolved conditions of r in the fact level.

3.3). The hybrid action types that have specified dynamics are represented by flow tubes. Fig. 4-8(b) shows an example of an action level. It also contains no-op actions.

An action type is instantiated in an action level, if its conditions are *resolved* in the previous fact level.

Definition 33. **[Resolved]** *A literal p is said to be* **resolved** *in fact level FL, if $\exists$ literal $p' \in FL$ such that $p = p'$. A continuous region r is said to be* **resolved** *in fact level FL, if $\exists r' \in FL$ such that $r \bigcap r' \neq \emptyset$. p' is called a* **resolved condition** *of p. r' is called a* **resolved condition** *of r. If p and r are conditions of an action a, then p' is called a* **discrete resolved condition** *of a, r' is called a* **continuous resolved condition** *of a, and the intersection of r and r', $r \bigcap r'$, is called a* **resolved intersection** *of a.*

When a literal is resolved in a fact level, there is only one resolved condition of this literal. However, when a continuous region is resolved in a fact level, there can be multiple resolved conditions of this continuous region. This is because many continuous regions can have nonempty intersection with a continuous region. For example, as shown in Fig. 4-9, suppose R_1, R_2 and R_3 are in a fact level. As R_1, R_2 and R_3 all have nonempty intersection with a region r, R_1, R_2 and R_3 are all resolved conditions of r in the fact level.

The concept of a discrete resolved condition is used in Graphplan. In a Planning

Graph, it is the literal in a fact level that matches a condition of an action. The concept of a continuous resolved condition is new. The pseudo code for deciding whether a literal or a continuous region is resolved is in Alg. 1. Line 1-8 states that if there exists a literal in the fact level that matches the condition, then the condition is resolved. Line 9-16 state that if there exists a continuous region that has a nonempty intersection with the condition, then the condition is resolved.

Alg. 1 Resolved(condition: *cond*, fact level: fl) **returns** *true* or $false$

```
1: if cond is a literal then
2:   for each literal p in fl do
3:     if cond = p then
4:       return true
5:     end if
6:   end for
7:   return false
8: end if
9: if cond is a continuous region then
10:  for each continuous region r in fl do
11:    if cond ⋂ r ≠ ∅ then
12:      return true
13:    end if
14:  end for
15:  return false
16: end if
```

Recall that in the previous section introducing flow tubes (Section 4.1), the initial region R_I is considered as given, in order to focus on the core concepts of flow tubes. Now I explain how the initial region R_I of a flow tube $ft(R_I, t)$ is derived.

When a continuous region r in fact level i has a non-empty intersection with the continuous condition of action a in action level i, in other words, when r is a resolved condition of a, the intersection is taken as the initial region R_I of the flow tube of action a. As shown in Fig. 4-10, suppose r represents the continuous region in a fact level, and suppose R_p represents the continuous condition of an action. They intersect, and the non-empty intersection is R_I, which is the initial region of the flow tube ft of the action.

When there are multiple such non-empty intersections for one action, multiple

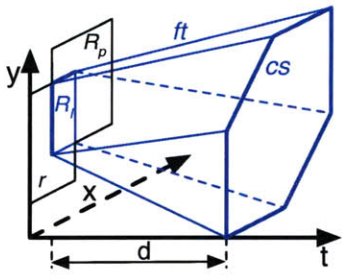


Figure 4-10: Suppose r represents the continuous region in a fact level, and suppose R_p represents the continuous condition of an action. They intersect, and the non-empty intersection is R_I, which is the initial region of the flow tube ft of the action.

flow tubes are constructed to instantiate this action. For example, in Fig. 4-8 there are two different flow tubes instantiating the action type `glide`. The flow tubes have different initial regions and end regions.

Recall that a hybrid action is called a *continuous action* if it has specified dynamics, otherwise it is called a *discrete action.* An action level is defined as follows.

Definition 34. [**Action Level**] *Given a fact level at $i \geq 0$, $FL(i)$, an action level at i, $AL(i)$, is $\langle DI, FT, noop \rangle$, where DI are discrete action instantiations whose conditions are resolved in $FL(i)$, FT are flow tubes for continuous actions whose conditions are resolved in $FL(i)$, and noop are no-op actions that preserve each of the propositional facts in $FL(i)$.*

After I described fact levels and action levels, I now show an example of a Hybrid Flow Graph, shown in Fig. 4-11. The Hybrid Flow Graph starts with *fact level* 1, followed by *action level* 1, *fact level* 2, and action level 2.

As shown in Fig. 4-11, each fact level contains continuous regions (in red) and literals (in black). Each action level contains hybrid actions. The continuous actions (in blue) are represented by flow tubes. Discrete actions are in black. Large black dots represent no-op actions. The flow tube of action `glide` in action level 1 is shown on top. Arrows connect resolved conditions to hybrid actions and hybrid actions to effects.

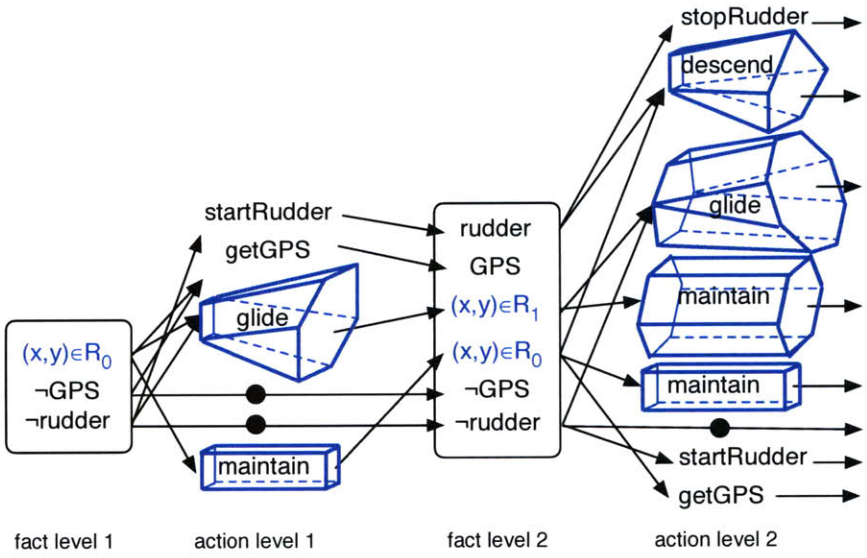


Figure 4-11: A Hybrid Flow Graph example. Each fact level contains continuous regions (in red) and literals (in black). Each action level contains hybrid actions. The continuous actions (in blue) are represented by flow tubes. Discrete actions are in black. Large black dots represent no-op actions. The flow tube of action `glide` in action level 1 is shown on top. Arrows connect resolved conditions to hybrid actions and hybrid actions to effects.

4.3 Hybrid Flow Graph Construction

Thus far I have introduced the flow tube representation for hybrid actions, as well as fact levels and action levels in a Hybrid Flow Graph. In this section, I use the building blocks to describe the construction of a Hybrid Flow Graph. Before presenting the graph expansion algorithm, I describe the mutual exclusion relations a Hybrid Flow Graph. They play an important role in pruning some of the invalid plans in the graph.

4.3.1 Mutual Exclusion

Two actions are *mutually exclusive* if no valid plan could contain both in the same action level. Likewise, two facts are *mutually exclusive* if no valid plan could possibly make both true in the same fact level. Similar to Graphplan, identifying and propagating *mutual exclusion* relations is an integral step in constructing the Hybrid Flow Graph. Identifying mutual exclusion relations can be largely useful in reducing the search for

a valid plan in a Hybrid Flow Graph [BF97].

To identify the mutual exclusion relations, Kongming propagates them forward from the first fact level throughout the Hybrid Flow Graph using a set of rules. The rules in Kongming are generalized from those in Graphplan, because in a Hybrid Flow Graph fact levels not only contain propositional facts but also continuous regions, and action levels contain hybrid actions instead of purely discrete actions. *Mutex* is used as a short form for mutual exclusion in the rest of this thesis.

Action Mutex

[Interference] In a given action level, two actions are mutex of each other, if a discrete effect of an action is the negation of a discrete condition or a discrete effect of the other action.

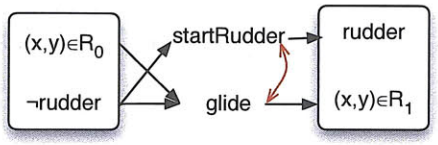


Figure 4-12: An example of action mutex based on the interference rule.

For example, in Fig. 4-12, `startRudder` and `glide` are mutex because an effect of `startRudder` negates a condition of `glide`. This rule only depends on the definition of action types, therefore, we know `startRudder` and `glide` are always mutex in any action level. This rule is essentially the same as in Graphplan, but it is more general as it applies to hybrid actions.

[Competing Needs] In a given action level, two actions are mutex of each other, if mutex relations exist between resolved conditions of two actions, in any of the following four cases:

1. Every continuous resolved condition of one action is mutex with every continuous resolved condition of the other action.

2. Every continuous resolved condition of one action is mutex with at least one discrete resolved condition of the other action.

3. At least one discrete resolved condition of one action is mutex with at least one discrete resolved condition of the other action.

4. Every resolved intersection of one action is mutex with every resolved intersection of the other action.

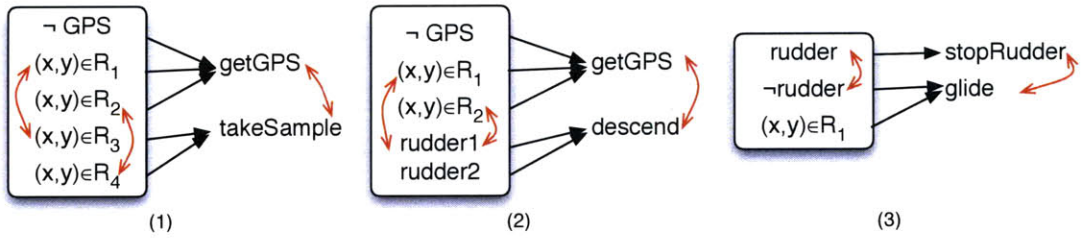


Figure 4-13: An example of action mutex for case 1-3 of the competing needs rule. (1) shows case 1. (2) shows case 2. (3) shows case 3.

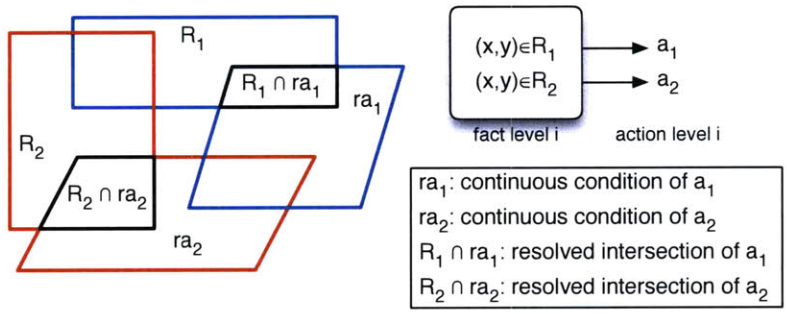


Figure 4-14: An example of action mutex for case 4 of the competing needs rule. The continuous condition of a_1 and the continuous condition of a_2 have a nonempty intersection, $ra_1 \cap ra_2 \neq \emptyset$. The resolved condition of a_1 and the resolved condition of a_2 have a nonempty intersection, $R_1 \cap R_2 \neq \emptyset$. However, the resolved intersection of a_1 and the resolved intersection of a_2 do not intersect, $(R_1 \cap ra_1) \cap (R_2 \cap ra_2) = \emptyset$. This prevents a_1 and a_2 from taking place at the same time.

Note that in the competing needs rule, continuous resolved conditions are treated differently from discrete resolved conditions. This is due to the reason discussed in 4.2.3; namely, in a fact level, a literal can only have one resolved condition, but a continuous region can have many resolved conditions.

Fig. 4-13 shows an example of each case of the competing needs rule. Suppose based on rules for mutex facts that will be described below, we have identified mutex pairs

in the fact level, connected with red arrows in the figure. (1) In the fact level, R_1 is mutex with R_3, and R_2 is mutex with R_4. Because every continuous resolved condition of `getGPS` is mutex with every continuous resolved condition of `takeSample`, `getGPS` and `takeSample` are mutex. (2) In the fact level, R_1 is mutex with `rudder1`, and R_2 is mutex with `rudder1`. Because every continuous resolved condition of `getGPS` is mutex with a discrete resolved condition of `descend`, `getGPS` and `descend` are mutex. (3) In the fact level, `rudder` and ¬`rudder` are mutex. Because a discrete resolved condition of `stopRudder` is mutex with a discrete resolved condition of `glide`, `stopRudder` and `glide` are mutex.

Fig. 4-14 shows an example of action mutex for case 4 of the competing needs rule. The continuous condition of a_1 and the continuous condition of a_2 have a nonempty intersection, $ra_1 \cap ra_2 \neq \emptyset$. The resolved condition of a_1 and the resolved condition of a_2 have a nonempty intersection, $R_1 \cap R_2 \neq \emptyset$. However, the resolved intersection of a_1 and the resolved intersection of a_2 do not intersect, $(R_1 \cap ra_1) \cap (R_2 \cap ra_2) = \emptyset$. This prevents a_1 and a_2 from taking place at the same time.

This rule not only depends on the definition of action types, but also depends on the mutex relations identified previously. This rule is more general than that in Graphplan, as the concept of a resolved condition is more general (Def. 33).

Fact Mutex

1. At a given fact level, two facts are mutex of each other, if they are literals, and one negates the other.

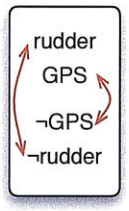


Figure 4-15: An example of fact mutex based on rule 1.

For example, in Fig. 4-15, `rudder` and ¬`rudder` are mutex because they negate

each other, and GPS and ¬GPS are mutex because they negate each other. This rule is the same as in Graphplan.

2. At a given fact level, two facts are mutex of each other, if they are continuous regions that do not intersect.

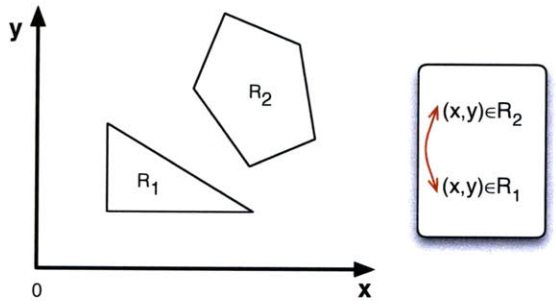


Figure 4-16: An example of fact mutex based on rule 2.

For example, in Fig. 4-16, suppose continuous region $R_1 = \{\mathbf{x} \in \mathcal{R}^n \mid \wedge_i f_i(\mathbf{x}) \leq 0\}$, and continuous region $R_2 = \{\mathbf{x} \in \mathcal{R}^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$. $R_1 \bigcap R_2 = \emptyset$, so there is no assignment to $\mathbf{x}$ such that both $\wedge_i f_i(\mathbf{x}) \leq 0$ and $\wedge_j g_j(\mathbf{x}) \leq 0$ are satisfied. Therefore, the state variables cannot be constrained to be in both continuous regions. This rule does not exist in Graphplan.

3. If all actions that create one fact are mutex with all actions that create the other fact.

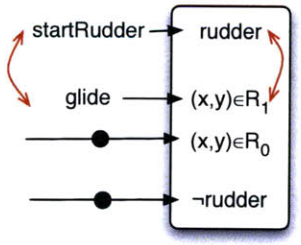


Figure 4-17: An example of fact mutex based on rule 3.

For example, in Fig. 4-17, in the fact level, `rudder` and $(x, y) \in R_1$ are mutex, because `startRudder` in the action level is the only way of creating `rudder`, and

`glide` is the only way of creating $(x, y) \in R_1$, and we know from the Interference rule example that `startRudder` and `glide` in the action level are mutex. This rule is essentially the same as in Graphplan, but it is more general as it applies to not only propositional facts but also continuous regions.

Two Actions Are Mutex:	
Interference	If a discrete effect of an action is the negation of a discrete condition or a discrete effect of the other action.
Competing Needs	1. Every continuous resolved condition of one action is mutex with every continuous resolved condition of the other action. 2. Every continuous resolved condition of one action is mutex with at least one discrete resolved condition of the other action. 3. At least one discrete resolved condition of one action is mutex with at least one discrete resolved condition of the other action. 4. Every resolved intersection of one action is mutex with every resolved intersection of the other action.

Table 4.1: Table of action mutex rules.

The procedure for identifying action mutex is in Alg. 2. It corresponds to the rules in Table 4.1. Line 2 corresponds to the Interference rule. Line 4 corresponds to the Competing Needs rule case 1, line 6 corresponds to case 2, line 8 corresponds to case 3, and line 10 corresponds to case 4.

The procedure for identifying fact mutex is in Alg. 3. It corresponds to the rules in Table 4.2. Line 2 corresponds to rule 1, line 4 corresponds to rule 2, and line 6 corresponds to rule 3.

	Two Facts Are Mutex:
1	If they are literals, and one negates the other.
2	If they are continuous regions that do not intersect.
3	If all actions that create one fact are mutex with all actions that create the other.

Table 4.2: Table of fact mutex rules.

Alg. 2 IdentifyActionMutex(action level k: $al(k)$, mutex fact pairs for level k: **fPairs**(k)) **returns** mutex action pairs for level k: **aPairs**(k). (a.eff: an effect of a; a.dp: a discrete condition of a; a.drp: a discrete resolved condition of a; a.crp: the set of all continuous resolved conditions of a; a.ri: the set of all resolved intersections of a.)

1: **for** each pair of actions $\langle a_1, a_2 \rangle$ in $al(k)$ **do**
2: **if** ($\exists\, a_1$.eff s.t. a_1.eff $= \neg a_2$.dp $\parallel$ a_1.eff $= \neg a_2$.eff) $\parallel$ ($\exists\, a_2$.eff s.t. a_2.eff $= \neg a_1$.dp $\parallel$ a_2.eff $= \neg a_1$.eff) **then**
3: **aPairs**(k).add($\langle a_1, a_2 \rangle$)
4: **else if** $\nexists\, r_1 \in a_1$.crp, $r_2 \in a_2$.crp s.t. $\neg \langle r_1, r_2 \rangle \in$ **fPairs**(k) **then**
5: **aPairs**(k).add($\langle a_1, a_2 \rangle$)
6: **else if** ($\exists\, a_1$.drp s.t. $\nexists\, r_2 \in a_2$.crp s.t. $\neg \langle a_1.\text{drp}, r_2 \rangle \in$ **fPairs**(k)) | ($\exists\, a_2$.drp s.t. $\nexists\, r_1 \in a_1$.crp s.t. $\neg \langle a_2.\text{drp}, r_1 \rangle \in$ **fPairs**(k)) **then**
7: **aPairs**(k).add($\langle a_1, a_2 \rangle$)
8: **else if** $\exists\, a_1$.drp, a_2.drp s.t. $\langle a_1.\text{drp}, a_2.\text{drp} \rangle \in$ **fPairs**(k) **then**
9: **aPairs**(k).add($\langle a_1, a_2 \rangle$)
10: **else if** $\nexists\, r_1 \in a_1$.ri, $r_2 \in a_2$.ri s.t. $\neg \langle r_1, r_2 \rangle \in$ **fPairs**(k) **then**
11: **aPairs**(k).add($\langle a_1, a_2 \rangle$)
12: **end if**
13: **end for**
14: **return aPairs**(k)

Alg. 3 IdentifyFactMutex(fact level k: $fl(k)$, mutex action pairs for level $k-1$: **aPairs**($k-1$)) **returns** mutex fact pairs for level k: **fPairs**(k). (a.rg: the resolved end region of the flow tube of a; a.noop: fact preserved by no-op action a.)

1: **for** each pair of facts $\langle f_1, f_2 \rangle$ in $fl(k)$ **do**
2: **if** f_1 is literal & f_2 is literal & $f_1 = \neg f_2$ **then**
3: **fPairs**(k).add($\langle f_1, f_2 \rangle$)
4: **else if** f_1 is continuous region & f_2 is continuous region & $f_1 \bigcap f_2 = \emptyset$ **then**
5: **fPairs**(k).add($\langle f_1, f_2 \rangle$)
6: **else if** $\langle a_1, a_2 \rangle \in$ **aPairs**($k-1$), $\forall \langle a_1, a_2 \rangle$ s.t. (a_1.eff $= f_1 \parallel a_1$.rg $= f_1 \parallel a_1$.noop $= f_1$) & (a_2.eff $= f_2 \parallel a_2$.rg $= f_2 \parallel a_2$.noop $= f_2$) **then**
7: **fPairs**(k).add($\langle f_1, f_2 \rangle$)
8: **end if**
9: **end for**
10: **return fPairs**(k)

4.3.2 Definition and Properties of A Hybrid Flow Graph

In this section, I formally define a Hybrid Flow Graph and its properties. A Hybrid Flow Graph is a leveled, directed graph, formally defined as in Def. 35.

Definition 35. [Hybrid Flow Graph] *A Hybrid Flow Graph of N levels is a 4-tuple* $\langle \mathbf{FL}, \mathbf{AL}, \mathbf{FM}, \mathbf{AM} \rangle$, *where*

- $\mathbf{FL} = \{FL(1), FL(2), \ldots, FL(N)\}$ *is a sequence of fact levels, each defined in Def. 32.* $FL(1) = \mathcal{I}$, *where* $\mathcal{I}$ *is the initial conditions (Def. 4).*
- $\mathbf{AL} = \{AL(1), AL(2), \ldots, AL(N-1)\}$ *is a sequence of action levels, each defined in Def. 34.*
- $\mathbf{FM} = \{FM(1), FM(2), \ldots, FM(N)\}$ *is a sequence of fact mutex sets.* $FM(i)$ *is a set of all pairs of mutex facts in fact level* i. *Each pair of mutex facts follows at least one rule in Table 4.2.*
- $\mathbf{AM} = \{AM(1), AM(2), \ldots, AM(N-1)\}$ *is a sequence of action mutex sets.* $AM(i)$ *is a set of all pairs of mutex actions in action level* i. *Each pair of mutex actions follows at least one rule in Table 4.1.*

A Hybrid Flow Graph represents the following.

- All *valid hybrid plans* (Def.36);
- Some invalid plans.

A *valid hybrid plan* is defined as follows.

Definition 36. [Valid Hybrid Plan] *A valid hybrid plan of N levels is a triple,* $\mathcal{P}_h = < \mathbf{A}, \mathbf{S}, \mathbf{U} >$, *where*

- $\mathbf{A} = \{\mathbf{a}(1), \mathbf{a}(2), \ldots, \mathbf{a}(N-1)\}$ *is a sequence of actions instantiated from the hybrid atomic action types* $\mathcal{AA}$ *(Def. 23).* $\mathbf{a}(i)$ *is the set of actions to be performed at level* i;

- $\mathbf{S} = \{\mathbf{s}(1), \mathbf{s}(2), \ldots, \mathbf{s}(N)\}$ *is a sequence of assignment to the continuous and discrete state variables* $\mathbf{s} = \langle \mathbf{x}, \mathbf{p} \rangle$ *(Def. 2).* $\mathbf{s}(i)$ *is the assignment to* $\mathbf{s}$ *at level* i*;*

- $\mathbf{U} = \{\mathbf{u}(1), \mathbf{u}(2), \ldots, \mathbf{u}(N-1)\}$ *is a sequence of assignment to the control variables* $\mathbf{u}$ *(Def. 3).* $\mathbf{u}(i)$ *is the assignment to* $\mathbf{u}$ *at level* i*.*

such that all of the following are true:

- *No actions at the same level are mutex.* $\forall i$ $\mathbf{a}(i)$ *contains no action pairs that are mutex (Table 4.1);*

- *If an action is performed at level* i*, then all its preconditions are resolved (Def. 33) at level* i*;*

- *If a fact is true at level* i*,* $i > 1$*, then at least one action that causes the fact to be true is performed at level* $i - 1$*;*

- *Each action in the action sequence satisfies its dynamics.* $\forall i = 1, \ldots, N-1$, $\mathbf{u}(i) \in Lim(a(i))$, $\forall a(i) \in \mathbf{a}(i)$, *and* $\{\mathbf{x}(i), \mathbf{u}(i), \mathbf{x}(i+1)\}$ *satisfy* $Tran_S(a(i))$, $\forall a(i) \in \mathbf{a}(i)$, *where* $Lim(a(i))$ *is the dynamic limitation of action* a *that takes place at level* i*, and* $Tran_S(a(i))$ *is the state transition equation of action* a *that takes place at level* i*;*

- *Initial conditions are satisfied.* $\mathbf{s}(1) = \langle \mathbf{x}(1), \mathbf{p}(1) \rangle \in \mathcal{I}$*, where* $\mathcal{I}$ *is the initial conditions (Def. 4);*

- *Goal conditions are satisfied.* $\mathbf{s}(N) = \langle \mathbf{x}(N), \mathbf{p}(N) \rangle \in \mathcal{G}$*, where* $\mathcal{G}$ *is the goal conditions (Def. 20);*

- *External constraints are satisfied.* $\forall c_i \in \mathcal{C}$*, the continuous state variables* $\{\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(N)\}$ *satisfy* c_i*, where* $\mathcal{C}$ *is the external constraints (Def. 16).*

I show an example of a valid hybrid plan in a Hybrid Flow Graph in Fig. 4-18. The simple planning task is to start from an initial point, take a sample from a specific region, and return to the starting point. To simplify the problem, there is no external constraints.

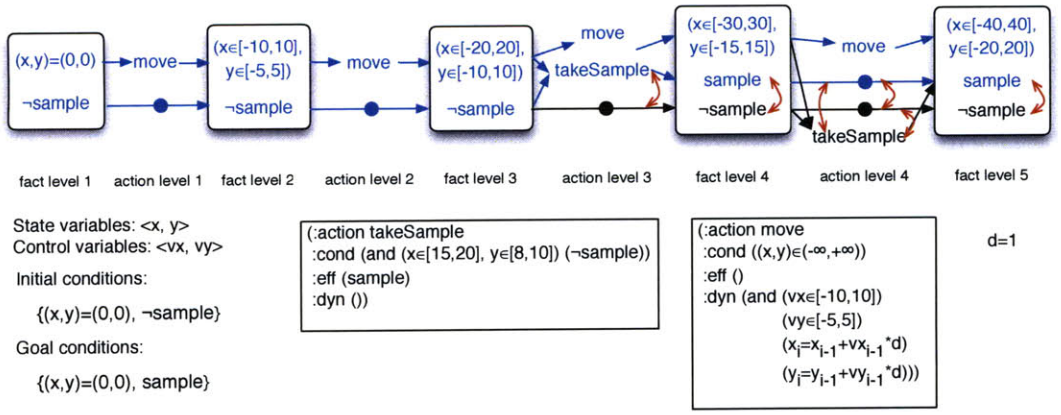


Figure 4-18: A Hybrid Flow Graph constructed for the planning problem listed below. Large dots represent no-op actions. Mutex relations are marked with red arrows. The sequence of concurrent actions and facts in blue shows a valid hybrid plan.

The details are listed in Fig. 4-18, and described as follows. Suppose the state variables are $< x, y >$ and control variables are $< vx, vy >$. The initial conditions are $\{(x, y) = (0, 0), \neg$ `sample`$\}$. The goal conditions are $\{(x, y) = (0, 0),$ `sample`$\}$. There are two action types: `takeSample` and `move`. Action type `takeSample` has its conditions being $\{(x \in [15, 20], y \in [8, 10]), \neg$ `sample`$\}$, and its effect being `sample`. `takeSample` has no dynamics. Action type `move` has its condition being $(x, y) \in (-\infty, +\infty)$, its actuation limits being $vx \in [-10, 10], vy \in [-5, 5]$, and its state equation being $x(t_i) = x(t_{i-1}) + vx(t_{i-1}) * d$, $y(t_i) = y(t_{i-1}) + vy(t_{i-1}) * d$. The fixed and equal duration over all actions is $d = 1$.

The valid plan consists of the following. `move` at time 1, `move` at time 2, `takeSample` and `move` at time 3, `move` at time 4. A consistent assignment to the state variables is $(x(1), y(1)) = (0, 0)$, $(x(2), y(2)) = (10, 5)$, $(x(3), y(3)) = (20, 10)$, $(x(4), y(4)) = (10, 5)$, $(x(5), y(5)) = (0, 0)$. A consistent assignment to the control variables is $(vx(1), vy(1)) = (10, 5)$, $(vx(2), vy(2)) = (10, 5)$, $(vx(3), vy(3)) = (-10, -5)$, $(vx(4), vy(4)) = (-10, -5)$.

Similar to the fact that a Planning Graph represents not only all valid plans but also some invalid plans, a Hybrid Flow Graph also represents some invalid plans. For example, the sequence of concurrent actions and facts in red in Fig. 4-19 shows an invalid plan, for the same planning problem shown in Fig. 4-18.

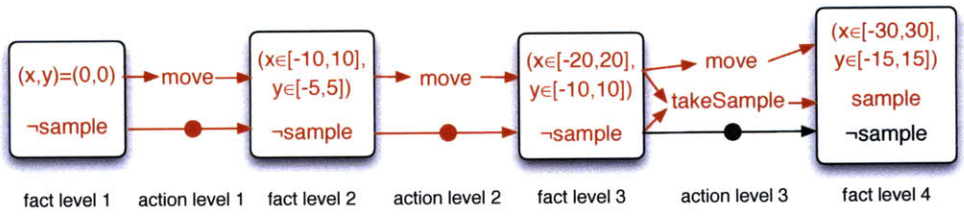


Figure 4-19:

To differ from the invalid plans that exist in a Planning Graph, this invalid plan example focuses on the inconsistent assignment to state and control variables. The goal conditions are contained in fact level 4, as $(x \in [-30, 30], y \in [-15, 15]) \bigcap (x = 0, y = 0) \neq \emptyset$ and `sample` is in the fact level, without being mutex. However, there exists no consistent assignment to $(x(3), y(3))$, $(x(4), y(4))$, and $(vx(3), vy(3))$, such that the continuous condition of `takeSample` at time 3 is satisfied, the dynamics of `move` at time 3 are satisfied, and $(x(4), y(4)) = (0, 0)$.

4.3.3 Defining *Contained*

Before introducing the graph expansion algorithm in $\mathrm{K_{AA}}$, I first introduce a simple concept, "contained", which will be useful in explaining the expansion algorithm in the following section. $\mathrm{K_{AA}}$ adds an action instantiation to an action level, if the conditions of the action are *contained* in the previous fact level. Intuitively, *contained* means *resolved* and without mutex relations.

Definition 37. [Contained] *Given a set of conditions,* $Cond = \{q_1, q_2, \ldots, q_m, r\}$, *where each* q_i *is a literal, and* r *is a continuous region,* $r = \{\mathbf{x} \in \mathcal{R}^n \mid \wedge_j g_j(\mathbf{x}) \leq 0\}$, *with each* $g_j(\mathbf{x})$ *being a linear function of* $\mathbf{x}$. *Given a fact level,* FL. *Given the complete set of mutex fact pairs in* FL, $fPairs$. $Cond$ *is contained in* FL, *if all of*

the following hold:

(*a*) $\forall i$, q_i *is resolved in* FL.

(*b*) r *is resolved in* FL.

(*c*) *Let* $R' = \{r'_1, r'_2, \ldots\}$ *be the set of all resolved conditions of* r. *Let* $Q' = \{q'_1, q'_2, \ldots, q'_m\}$ *be the set of resolved conditions of* $\{q_1, q_2, \ldots, q_m\}$. *Then* $\exists\ r' \in R'$ *s.t.* $\nexists\ f_1, f_2 \in \{Q' \bigcup r'\}$ *s.t.* $< f_1, f_2 > \in fPairs$.

Def. 37(c) states two requirements. First, at least one of the resolved conditions of r is not mutex with any of the resolved conditions of q_i. Second, no two of the resolved conditions of q_i are mutex. The pseudo code for checking whether specific conditions are *contained* in a fact level is in Alg. 4. Line 1-6 correspond to Def. 37(a), line 7-10 correspond to Def. 37(b), and line 11-13 correspond to Def. 37(c). The pseudo code for Resolved() is Alg. 1.

Alg. 4 Contained(conditions: $\{q_1, q_2, \ldots, q_m, r\}$, fact level: fl, mutex fact pairs in fl: $fPairs$) **returns** *true* or *false*

```
1:  for each q_i do
2:     if Resolved(q_i,fl) then
3:        a ← true
4:        q'_i ← resolved condition of q_i
5:     end if
6:  end for
7:  if Resolved(r,fl) then
8:     b ← true
9:     R' = {r'_1, r'_2, ...} ← all resolved conditions of r
10: end if
11: if ∃ r' ∈ R' s.t. ∄ f_1, f_2 ∈ {q'_1, q'_2, ..., q'_m, r'} s.t. < f_1, f_2 > ∈ fPairs then
12:    c ← true
13: end if
14: if a & b & c then
15:    return true
16: else
17:    return false
18: end if
```

For example, as shown in Fig. 4-20, fact level i is on the left, and the set of

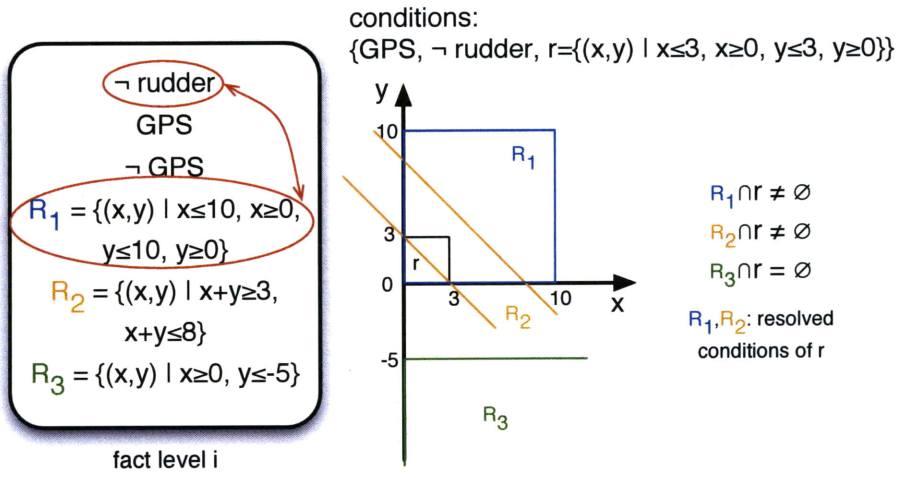


Figure 4-20: Fact level i is on the left, and the set of conditions are listed on the top. Suppose ¬`rudder` and R_1 in the fact level are known to be mutex, connected by red arrows. Conditions `GPS` and ¬`rudder` are resolved in the fact level. Condition r is resolved. Both R_1 and R_2 are resolved conditions of r. `GPS` and ¬`rudder` are not mutex, and R_2 is not mutex with either `GPS` or ¬`rudder` in the fact level. Therefore, the conditions are contained in the fact level.

conditions are listed on the top. Suppose ¬`rudder` and R_1 in the fact level are known to be mutex, connected by red arrows. Alg. 4 line 1-6 check literals: we know `GPS` and ¬`rudder` are resolved in the fact level. Alg. 4 line 7-10 check the continuous region: we know r is resolved. Both R_1 and R_2 are resolved conditions of r. Alg. 4 line 11-13 check mutex: we know `GPS` and ¬`rudder` are not mutex, and R_2 is not mutex with either `GPS` or ¬`rudder` in the fact level. Therefore, the conditions are contained in the fact level.

4.3.4 Graph Expansion Algorithm

Now we are ready to introduce the algorithm for expanding the Hybrid Flow Graph, ExpandGraph(). The input of ExpandGraph() is a subset of the input of K_{AA}, including initial conditions, final goal conditions, hybrid atomic action types, and external constraints that are unit clauses. The output of ExpandGraph() is a Hybrid Flow Graph.

On the high level, the initial conditions of the hybrid planning problem form the first fact level of the Hybrid Flow Graph. The first fact level is followed by the first

action level, which consists of all the actions whose conditions are *contained* in the first fact level. Then the effects of these actions form the next fact level. The Hybrid Flow Graph keeps expanding until the goal conditions are *contained* in a fact level. The pseudo code is in Alg. 5. It consists of the following steps:

Alg. 5 ExpandGraph(hybrid action types: $aSet$, initial conditions: $\mathcal{I}$, goal conditions: $\mathcal{G}$, unit-clause external constraint set: $\mathcal{C}'$) **returns** Hybrid Flow Graph: hfg

```
1: hfg(1) ← Initialize(I) {Alg. 6}
2: k ← 1
3: while !Contained(G, fl(k), fPairs(k)) & k ≤ N do
4:    hfg(k + 1) ← ExpandOneLevel(hfg(k), aSet, C') {Alg. 7}
5:    k ← k + 1
6: end while
7: return hfg
```

- Initialize (line 1). It is described in Alg. 6.
- While the goal conditions are not *contained* in the current fact level within N iterations, repeat the process of expanding the graph by one level (line 3-6). ExpandOneLevel() is described in Alg. 7. This is limited to N iterations in order to avoid running into an infinite loop when the goal conditions are not reachable. This is discussed in Section 4.3.5.

Alg. 6 Initialize(initial conditions: $\mathcal{I}$) **returns** 1-level Hybrid Flow Graph: $hfg(1)$

```
1: hfg ← {} {create an empty hybrid flow graph.}
2: fl(1) ← I {create the first fact level from the initial conditions.}
3: fPairs(1) ← {} {create an empty fact mutex set for fact level 1.}
4: hfg.add(fl(1)), hfg.add(fPairs(1))
5: return hfg
```

In Initialize() (Alg. 6), K_{AA} creates fact level 1 from the initial conditions, initializes the fact mutex set for fact level 1 as empty, and adds the fact level and fact mutex set to an empty Hybrid Flow Graph. The mutex set for fact level 1 is empty, because it is assumed that the initial conditions specified in the input are not mutex. For example, as shown in Fig. 4-21, the initial conditions form fact level 1.

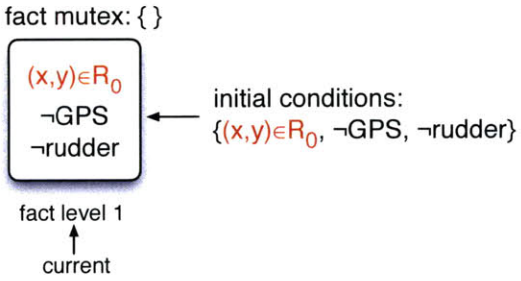


Figure 4-21: Initialization: K_{AA} creates fact level 1 from the initial conditions, initializes the fact mutex set for fact level 1 as empty.

Alg. 7 ExpandOneLevel(k-level Hybrid Flow Graph: $hfg(k)$, hybrid action types: $aSet$, unit-clause external constraint set: $\mathcal{C}'$) **returns** $(k+1)$-level Hybrid Flow Graph: $hfg(k+1)$

---

if $k \geq 2$ **then**
  $fPairs(k) \leftarrow$ IdentifyFactMutex($fl(k)$, $aPairs(k-1)$) {Alg. 3}
  hfg.add($fPairs(k)$)
end if
$al(k) \leftarrow \{\}$, $fl(k+1) \leftarrow \{\}$ {create the current action level and the next fact level, both initialized empty.}
for each action type a in $aSet$ **do**
  $\langle al(k), fl(k+1)\rangle \leftarrow$ InsertAction(a, $fl(k)$, $fPairs(k)$, d, $\mathcal{C}'$, $al(k)$, $fl(k+1)$) {Alg. 8}
end for
for each discrete fact f in $fl(k)$ **do**
  if f does not contain *-int* suffix **then**
    $al(k)$.add(new no-op)
    $fl(k+1)$.add(f)
  end if
end for
$aPairs(k) \leftarrow$ IdentifyActionMutex($al(k)$, $fPairs(k)$) {Alg. 2}
hfg.add($al(k)$), hfg.add($fl(k+1)$), hfg.add($aPairs(k)$)
return hfg

The algorithm for expanding the graph by one level is in Alg. 7. It consists of the following steps.

- If there are two or more levels in the graph, then identify fact mutex in the current fact level and add the fact mutex set to the graph (line 1-4). The algorithm for finding fact mutex is in Alg. 3. The rules were explained in Section 4.3.1. Because the fact mutex set for fact level 1 is empty and is already included in the graph in Initialize(), fact level 1 is skipped here.

- Check whether to instantiate each hybrid action type in the current action level (line 6-8). The algorithm InsertAction() is described in Alg. 8.

- Create a no-op action for each literal in the current fact level, unless the literal contains a "-int" suffix, and add the facts to the next fact level (line 9-14). Checking the "-int" suffix is a feature specifically created for K_{DA}. It will be explained in Chapter 6.

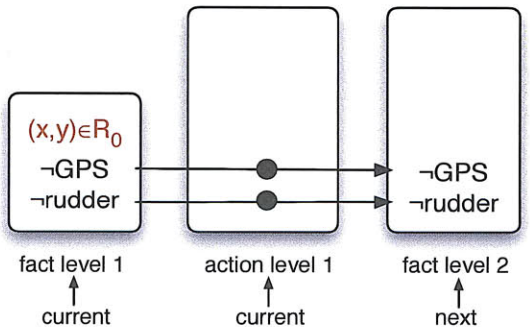


Figure 4-22: No-op actions are created for literal ¬`rudder` and literal ¬`GPS` in fact level 1, represented by black dots.

For example, in Fig.4-22, no-op actions are created for literal ¬`rudder` and literal ¬`GPS` in fact level 1, represented by black dots in the figure. The no-op actions are added to action level 1, and the literals they preserve are added to fact level 2.

- Identify action mutex in the current action level (line 13). The algorithm for finding action mutex is in Alg. 2. The rules were explained in Section 4.3.1.

- Add the action level and the next fact level to the Hybrid Flow Graph (line 14).

Alg. 8 InsertAction(a hybrid action: a, current fact level: fl, fact mutex pairs of the fact level: $fPairs$, duration: d, unit-clause external constraints: $\mathcal{C}'$, current action level: al, next fact level: nfl) **returns** current action level and next fact level: $\langle al, nfl \rangle$

```
1: if Contained(a.conditions, fl, fPairs) then
2:   if a.dynamics = ∅ then
3:     al.add(a)
4:   else
5:     for each crp in a's continuous resolved conditions do
6:       resolved end region rg ← ComputeFlowTube(a, crp, d) ⋂ C'
7:       if rg ≠ ∅ then
8:         al.add(a)
9:         nfl.add(rg)
10:      end if
11:    end for
12:  end if
13:  nfl.add(a.effects)
14: end if
15: return ⟨al, nfl⟩
```

Algorithm InsertAction() checks a hybrid action to see whether it should be added to the action level, as shown in Alg. 8. The algorithm can be divided into the following three parts. They are explained with the example shown in Fig.4-23.

- If the conditions of a hybrid action type are *contained* in the current fact level, then this action type is instantiated (line 1).

  For example, in Fig.4-23, the conditions of `startRudder`, `getGPS` and `glide` are all *contained* in fact level 1.

- If this action type has no dynamics (line 2), the instantiation is simple. Namely, the action is added to the current action level (line 3), and its discrete effects are added to the next fact level (line 13).

  For example, in Fig.4-23, `startRudder` and `getGPS` have no dynamics. They are added to action level 1, and their discrete effects are added to fact level 2.

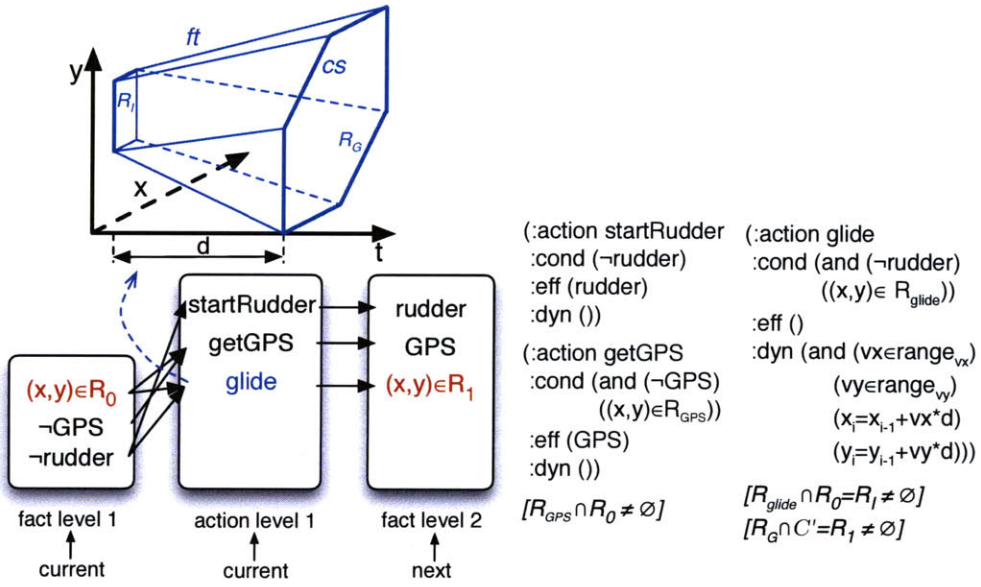


Figure 4-23: The action types are defined on the right. The conditions of `startRudder`, `getGPS` and `glide` are all *contained* in fact level 1. `startRudder` and `getGPS` have no dynamics. They are added to action level 1, and their discrete effects are added to fact level 2. `glide` has dynamics. There is one continuous resolved condition, $(x, y) \in R_0$, so one flow tube is constructed, shown on the top. The initial region of the flow tube, R_I, is the intersection of the continuous condition of `glide`, R_{glide}, and the continuous resolved condition of `glide`, R_0. The resolved end region of the flow tube, R_1, is the intersection of the end region of the flow tube, R_G, and the unit-clause external constraints, $\mathcal{C}'$. R_1 is not empty, and is added to fact level 2.

- If this action type has dynamics, a flow tube is computed for each continuous resolved condition of this action (line 5-11). Flow tube computation is as described in Section 4.1. The resolved end region (Def. 31) is the intersection of the end region of the flow tube and the unit-clause external constraints (line 6). If the resolved end region is not empty (line 7), the flow tube is added to the current action level (line 8), and its resolved end region is added to the next fact level (line 9). Its discrete effects are added to the next fact level (line 13).

  For example, in Fig.4-23, `glide` has dynamics. There is one continuous resolved condition, $(x, y) \in R_0$, so one flow tube is constructed. The initial region of the flow tube is the intersection of the continuous condition of `glide` and the continuous resolved condition of `glide`. The resolved end region of the flow tube is the intersection of the end region of the flow tube and the unit-clause external constraints. The resolved end region is not empty, and is added to fact level 2.

4.3.5 Level Off

When there exists no solution to the planning problem, there should be a mechanism to prevent the planning algorithm from running forever through an infinite number of stages. Graphplan is guaranteed to return failure when no solution exists [BF97], due to its "level off" property. I first review the definition of "level off" and the reasoning behind it. I then discuss the termination conditions of $\mathrm{K_{AA}}$ when no solution exists.

A Planning Graph is said to have *leveled off*, when it reaches the point where two adjacent proposition levels P_n and P_{n+1} have identical propositions and mutex relations. It is called "level off", because as shown in [BF97], once two adjacent proposition levels P_n and P_{n+1} are identical, all future levels will be identical to P_n as well. Intuitively, this is because first, propositions, once added to a level, remain in successive levels. In other words, propositions monotonically increase. Second, once a mutex has decayed, it never reappears. In other words, mutex relations monotonically decrease. Hence, the graph eventually reaches a fix point, where propositions and mutex relations no longer change.

In Graphplan, there are two termination tests for when no solution exists. First, if the goal conditions are not *contained* in the "level off" level P_n, then the goal conditions will not be contained in any levels after P_n either. In this case, Graphplan returns failure. Second, if the goal conditions are contained, but higher order exclusions, called *memos*, are preventing a solution. *Memos* are exclusions higher order than mutual exclusions, which also monotonically decrease. They may change after the "level off" point but will eventually reach a fixed point. Hence, the second termination test is to test whether a solution can be extracted after memos stop changing.

In K_{AA}, however, the Hybrid Flow Graph is not guaranteed to level off, due to the continuous regions in fact levels. More specifically, there are two reasons. First, recall from Section 4.2.1, that no-op actions do not preserve continuous states from one fact level to the next. If the system under control cannot perform the "maintain" action, then continuous regions do not monotonically increase in fact levels, in contrast to the propositions in a Planning Graph. The second reason is that unlike the propositions predefined in the input, there is an unlimited collection of continuous regions that can be contained in fact levels. It is possible that new continuous regions are created in every fact level as time elapses. Therefore, the graph is not guaranteed to reach a fixed point, even if facts and actions were monotonically increasing. In the special case, however, where Kongming is limited to purely discrete actions and propositional facts, as in Graphplan, Kongming follows the same termination conditions as in Graphplan, and Kongming is guaranteed to return failure when no solution exists.

In this chapter, I introduced the flow tube representation of hybrid actions, the Hybrid Flow Graph to represent all valid plans, and the algorithm for constructing a Hybrid Flow Graph. In the next chapter, I will introduce the constraint-based planning algorithm that K_{AA} employs on the Hybrid Flow Graph.

Chapter 5

Planning Algorithm for K_{AA}

Contents

Recall that K_{AA} consists of two parts: a compact representation of the space of possible hybrid plans and a constraint-based planning algorithm that operates on the hybrid plan representation. I introduced the plan representation in Chapter 4. In this chapter, I introduce the constraint-based planning algorithm.

For discrete planning problems, Planning Graphs [BF97] have been employed within a range of planning paradigms. The most notable planning paradigms are constraint-based planners and heuristic search planners. Constraint-based planners [WW99, SD05, KS92, KS99, DK01a] formulate the discrete planning problem as a satisfiability (SAT) problem or constraint satisfaction problem (CSP), and solve it using a SAT or CSP solver. The strength of constraint-based planners is that they leverage the advantage of state-of-the-art constraint solvers to perform the search for a valid plan. Heuristic forward search planners [CCFL09, CFLS08a, HN01, MBB+09,

DK01b, BG99, BG01, McD96] perform search directly on the plan representation using heuristics obtained from relaxation. The strength of heuristic search planners is that heuristics estimate the cost to goal to guide the search for a valid plan.

For hybrid planning problems, where actions have both continuous and discrete effects, K_{AA} employs a constraint-based planning algorithm on the Hybrid Flow Graph representation. I will discuss the heuristic forward search approach as future work in Chapter 9. The planning algorithm of K_{AA} takes an analogous approach to Blackbox [KS99], which operates on Planning Graphs. Blackbox encodes the Planning Graph as a SAT problem, and solves it using a state-of-the-art SAT solver. Kongming generalizes the Blackbox approach, and encodes the Hybrid Flow Graph as mixed logical quadratic programs (MLQPs). Kongming solves the MLQPs using a state-of-the-art solver, currently CPLEX.

This chapter is organized as follows. First, I review Blackbox. Second, I introduce the high-level algorithm in K_{AA}, in comparison with the high-level algorithm in Blackbox. Finally, I describe the constraint encoding of the Hybrid Flow Graph.

5.1 Blackbox

Blackbox [KS99] is a planning system that unifies the planning as satisfiability framework [KS92] with the Planning Graph approach [BF97]. Graphplan is good at instantiating the propositional structure, while SATPLAN uses powerful search algorithms [KS99]. Blackbox combines the advantages from both worlds: the instantiation power of Graphplan and the search capability from SAT solvers. Blackbox solves the planning problem in the following process.

1. Turns STRIPS input into a Planning Graph of length k. It interleaves instantiation and pruning using mutex.

2. Translates the Planning Graph into a conjunctive normal form (CNF) wff[1].

[1] A wff is a well-formed formula. It is an abstraction expressed using the symbols and formal grammar of a formal language.

3. Simplifies using general limited deduction, such as unit propagation, failed literal, binary failed literal.

4. Solves the SAT problem using a state-of-the-art solver.

5. If a model of the wff is found, then converts a model to the corresponding plan; otherwise, increments k and repeat the process.

The algorithm architecture of Blackbox is shown in the diagram in Fig. 5-1.

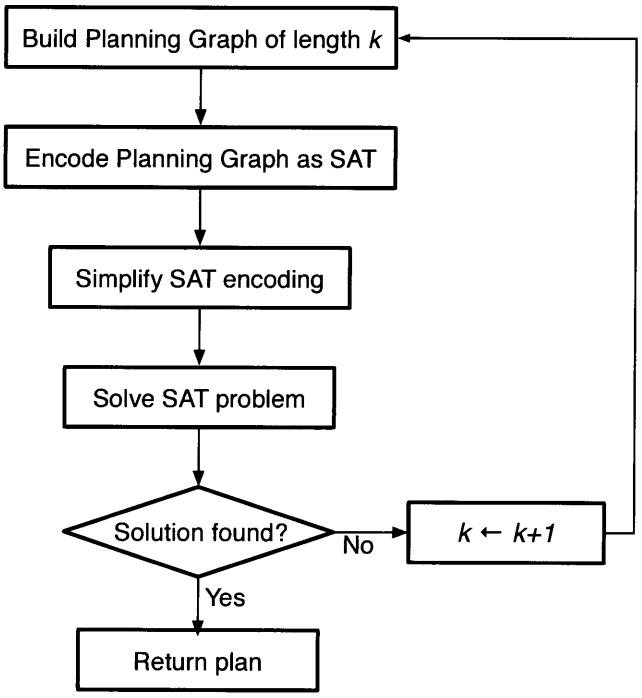


Figure 5-1: The algorithm architecture of Blackbox.

Blackbox represents each proposition and each action in a Planning Graph with a propositional variable p. $p = true$ if the corresponding propositional fact or action is included in the valid plan; otherwise $p = false$. Blackbox employs a few simple rules to translate the Planning Graph into a CNF wff. I explain them as follows, along with the example in Fig. 5-2.

- All facts in fact level 1 are true. This is because they are the initial conditions.

- The last fact level contains the goal conditions.

- A fact in level $i+1$ implies at least one of the actions that cause it in level i. For example, in Fig. 5-2, "Fact1" implies at least one of "Action1" and "Action2". $Fact1 \implies Action1 \vee Action2$.

- An action in level i implies all its conditions in level i. For example, in Fig. 5-2, "Action1" implies both of "Pre1" and "Pre2". $Action1 \implies Pre1 \wedge Pre2$.

- Mutex facts or actions cannot co-exist. For example, in Fig. 5-2, "Action2" and "Action3" cannot co-exist, and "Fact1" and "Fact2" cannot co-exist. $\neg Action2 \vee \neg Action3$, $\neg Fact1 \vee \neg Fact2$.

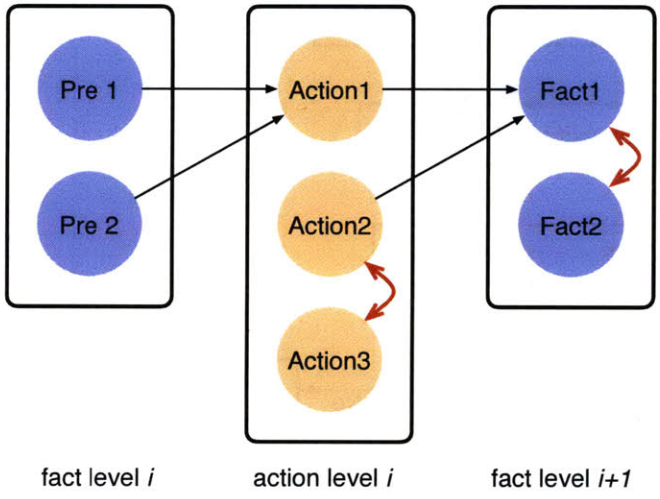


Figure 5-2: A Planning Graph example to demonstrate the encoding rules of Blackbox. "Pre1" and "Pre2" are conditions of "Action1". "Action1" and "Action2" both have "Fact1" as an effect. "Action2" and "Aciton3" are mutex. "Fact1" and "Fact2" are mutex.

5.2 K_{AA} Algorithm Architecture

I explain the high-level algorithm of K_{AA} in this section. Recall that the input to the K_{AA} is defined in Section 3.3. The input consists of initial conditions, goal conditions, hybrid atomic action types, external constraints and an objective function. The output of K_{AA} is an optimal plan, as defined in Def. 27.

The high-level algorithm for K_{AA} expands the Hybrid Flow Graph until the goal conditions are contained. It then interleaves between expanding the graph by one level (action and fact), and encoding the graph as a mixed logical quadratic program (MLQP) and solving it, until an optimal plan is found. The algorithm architecture is shown in the diagram in Fig. 5-3.

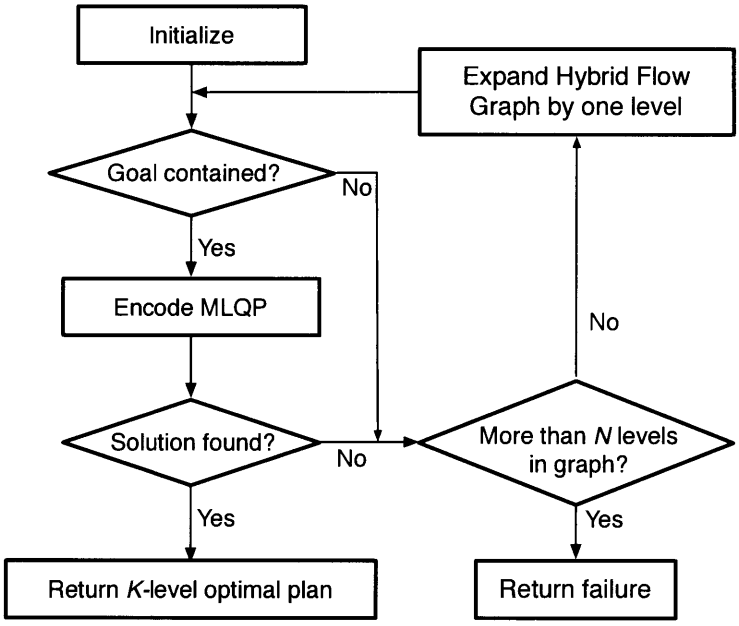


Figure 5-3: The algorithm for K_{AA}.

The pseudo code of this high-level algorithm is given by Alg. 9. I explain the pseudo code in accordance with the diagram in Fig. 5-3 as follows.

- Initialize (line 1). The Hybrid Flow Graph of 1 level is created through Initialize() (Alg. 6 in Chapter 4).

- Repeat until the number of levels in the graph reaches N (line 2-10). N is the cutoff number to prevent K_{AA} from running infinitely when no solution exists. Within the loop, check two conditions:

  - Check whether the goal conditions are contained in the last level of the graph (line 3). If they are, encode the graph as an MLQP and solve it (line 4). The algorithm for encoding the MLQP is in Alg. 10.

– Check whether there is a solution to the MLQP (line 5). If there is, return the solution (line 6).

- When there is no solution for the graph, expand the graph by one level (line 9). The algorithm for expanding the graph by one level is in Alg. 7 in Chapter 4.

Alg. 9 $\mathbf{K_{AA}}$(state variables: **x**, control variables: **u**, propositional variables for facts: **pf**, propositional variables for actions: **pa**, initial conditions: $\mathcal{I}$, goal conditions: $\mathcal{G}$, hybrid action types: $\mathcal{HT}$, external constraint set: $\mathcal{C}$, unit-clause external constraint set: $\mathcal{C}' \subseteq \mathcal{C}$, objective function: f) **returns** optimal plan: *solution*

```
1: hfg(1) ← Initialize(I) {Alg. 6}
2: for k ← 1: k ≤ N: k ← k + 1 do
3:   if Contained(G, fl(k), fPairs(k)) then
4:     solution ← EncodeMLQP&Solve(hfg(k), x, u, pf, pa, I, G, HT, C, f)
       {Alg. 10}
5:     if solution ≠ nil then
6:       return solution
7:     end if
8:   end if
9:   hfg(k + 1) ← ExpandOneLevel(hfg(k), HT, C') {Alg. 7}
10: end for
```

Compared with the high-level algorithm in Blackbox, $\mathrm{K_{AA}}$ is different in the following aspects.

- $\mathrm{K_{AA}}$ encodes mixed logical quadratic programs, whereas Blackbox encodes satisfiability formulas. This is because there are real-valued variables and linear constraints in $\mathrm{K_{AA}}$.

- Blackbox employs SAT simplification techniques on the encoding before giving it to a SAT solver, whereas $\mathrm{K_{AA}}$ does not simplify the encoding before giving it to the MLQP solver. This is because a set of well-developed inexpensive techniques exist for SAT formulas, but not for MLQP formulas. Developing such techniques is not the focus of this thesis.

- $\mathrm{K_{AA}}$ searches for a k-level optimal plan, whereas Blackbox searches for a valid plan. $\mathrm{K_{AA}}$ optimize the solution plan according to the objective function specified in the input.

- When no solution exists, K_{AA} checks whether the number of levels in the graph has exceeded N, as the termination test. Blackbox returns failure when no solution exists, as it implements Graphplan's termination test.

5.3 K_{AA} Constraint Encoding

In this section, I introduce the algorithm for encoding the Hybrid Flow Graph as an MLQP, namely, EncodeMLQP&Solve() as previously mentioned in Alg. 9.

The input to EncodeMLQP&Solve() is $\langle hfg(k), \mathbf{x}, \mathbf{u}, \mathbf{pf}, \mathbf{pa}, \mathcal{I}, \mathcal{G}, \mathcal{HT}, \mathcal{C}, f \rangle$, where $hfg(k)$ is a k-level Hybrid Flow Graph, $\mathbf{x}$ is the continuous state variables, $\mathbf{u}$ is the control variables, $\mathbf{pf}$ is the propositional variables for facts, $\mathbf{pa}$ is the propositional variables for actions, $\mathcal{I}$ is the initial conditions, $\mathcal{G}$ is the goal conditions, $\mathcal{HT}$ is the set of hybrid action types, $\mathcal{C}$ is the set of external constraints, and f is the objective function. The output of EncodeMLQP&Solve() is an optimal plan, as defined in Def. 27.

5.3.1 Mixed Logical Quadratic Program

I define an MLQP in this section. As my definition of an MLQP derives from that of an MLLP, introduced in [HO99], I review the definition of an MLLP first.

$$\begin{aligned} &Minimize\ cx \\ &Subject\ to\ \ p_j(y,h) \rightarrow (A^j x \geq a^j), j \in J \mid\ q_i(y,h), i \in I. \end{aligned} \tag{5.1}$$

In Eq. 5.1, $p_j(y,h)$ and $q_i(y,h)$ are general logic wff's. $y = (y_1, \ldots, y_n)$ are propositional variables. $h = (h_1, \ldots, h_m)$ are finite-domain variables. For example, $p_j(y,h)$ or $q_i(y,h)$ can be $(y_1 \vee y_2) \wedge (h_1 \neq h_2)$. $A^j x \geq a^j$ is a system of linear inequalities, where x are real-valued variables. $A^j x \geq a^j$ is enforced when $p_j(y,h)$ is true. The objective function cx is a linear function of x.

In this thesis, I modify the definition in Eq. 5.1 as follows.

- I generalize the objective function to allow quadratic functions. This is because

it is useful to minimize distance traveled, which is quadratic in the position variables. The objective is either a linear or a quadratic function of the real-valued variables $\mathbf{s}$, where $\mathbf{s} = \langle \mathbf{x}, \mathbf{u} \rangle$.

- I remove the finite-domain variables, because as mentioned before, ($\mathbf{x}$, $\mathbf{u}$, $\mathbf{pf}$, $\mathbf{pa}$) are the only variables in the input to EncodeMLQP&Solve(). ($\mathbf{x}$, $\mathbf{u}$) are real-valued variables, and ($\mathbf{pf}$, $\mathbf{pa}$) are propositional variables.

In this thesis, I define an MLQP as:

$$\begin{aligned} &Minimize \;\; f(\mathbf{s}) \\ &Subject\ to \;\; p_j(\mathbf{p}) \rightarrow (A^j\mathbf{s} \geq a^j), j \in J \mid \; q_i(\mathbf{p}), i \in I. \end{aligned} \tag{5.2}$$

where $\mathbf{s} = \langle \mathbf{x}, \mathbf{u} \rangle$, and $\mathbf{p} = \langle \mathbf{pf}, \mathbf{pa} \rangle$.

In Eq. 5.2, $f(\mathbf{s})$ is a linear or quadratic objective function of $\mathbf{s}$. $p_j(\mathbf{p})$ and $q_i(\mathbf{p})$ are general logic wff's over propositional variables, $\mathbf{p} = \langle \mathbf{pf}, \mathbf{pa} \rangle$. For example, $p_j(\mathbf{p})$ or $q_i(\mathbf{p})$ can be $(pf_1 \vee pa_1) \wedge \neg pa_3$. $A^j\mathbf{s} \geq a^j$ is a system of linear inequalities, where $\mathbf{s}$ are real-valued variables. $A^j\mathbf{s} \geq a^j$ is enforced when $p_j(\mathbf{p})$ is true.

5.3.2 Encoding Rules

A Hybrid Flow Graph is encoded into a MLQP. The specific encoding rules of $\mathrm{K_{AA}}$ are as follows, in accordance with Alg. 10.

1. All facts in fact level 1 are true (line 1). This is because fact level 1 is formed from the initial conditions $\mathcal{I}$. It is specified in Eq. 5.3.

$$\wedge pf_i(1), \; \forall pf_i(1) \in \mathbf{pf}(1) \tag{5.3}$$

   For example, in Fig. 5-4, the corresponding constraint encoding is $pf_1(1) \wedge pf_2(1) \wedge pf_3(1)$.

2. Facts that are true in the last fact level of the k-level Hybrid Flow Graph contain

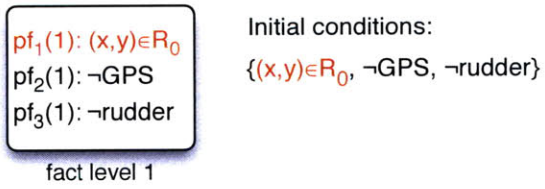


Figure 5-4: An example of the first level of a Hybrid Flow Graph. The initial conditions are listed on the right.

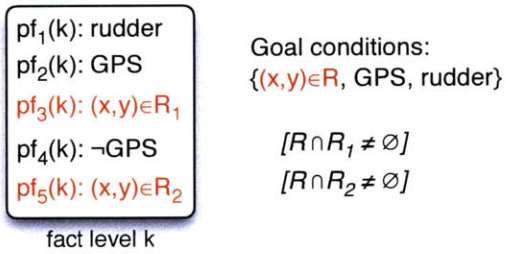


Figure 5-5: An example of the last level of a k-level Hybrid Flow Graph. The goal conditions are listed on the right.

the goal conditions $\mathcal{G}$ (line 2). It is specified in Eq. 5.4.

$$(\wedge pf_i(k),\ i \in I) \wedge (\vee pf_j(k),\ j \in J) \wedge (\mathbf{x} \in r_G), \tag{5.4}$$

where $pf_i(k),\ i \in I$ represent the discrete resolved conditions of $\mathcal{G}$ in level k, $pf_j(k),\ j \in J$ represent the continuous resolved conditions of $\mathcal{G}$ in level k, and r_G represents the continuous region in $\mathcal{G}$.

For example, in Fig. 5-5, the corresponding constraint encoding is $pf_1(k) \wedge pf_2(k) \wedge (pf_3(k) \vee pf_5(k)) \wedge (x, y) \in R$. As explained in Section 4.2.3, multiple continuous regions in a fact level can have nonempty intersections with a continuous region, but only one of them is needed in a valid plan.

3. If fact f in fact level m is true, then at least one of the actions in action level $m-1$ that have f either as a discrete effect or as a resolved goal region is true (line 3). It is specified in Eq. 5.5.

$$pf(m) \implies (\vee pa_i(m-1),\ i \in I), \tag{5.5}$$

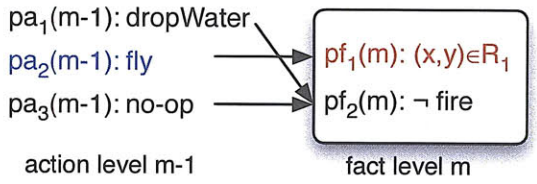


Figure 5-6: ¬`fire` is an effect of `dropWater` and the effect of an no-op action. $(x, y) \in R_1$ is the resolved goal region of `fly`.

where $pf(m)$ represents the fact f in fact level m, $pa_i(m-1),\ i \in I$ represent all the actions in action level $m-1$ that cause f.

For example, in Fig. 5-6, the corresponding constraint encoding is $pf_1(m) \implies pa_2(m-1)$, and $pf_2(m) \implies pa_1(m-1) \vee pa_3(m-1)$.

4. If action a in action level m is true, then (1) a's resolved conditions in fact level m are true, and (2) the state variables for level m satisfy a's continuous condition (line 4). They are specified in Eq. 5.6.

$$pa(m) \implies \begin{cases} (1)\ (\wedge pf_i(m),\ i \in I) \wedge (\vee pf_j(m),\ j \in J) \\ (2)\ \mathbf{x}(m) \in r \end{cases} \tag{5.6}$$

where $pf_i(m),\ i \in I$ represent the discrete resolved conditions of a, $pf_j(m),\ j \in J$ represent the continuous resolved conditions of a, and r represents the continuous condition of a.

For example, in Fig. 5-7, the corresponding constraint encoding is $pa_1(m) \implies pf_1(m) \wedge pf_2(m) \wedge (x, y) \in R_{fly}$, and $pa_2(m) \implies pf_3(m) \wedge (pf_1(m) \vee pf_4(m)) \wedge (x, y) \in R$.

5. Mutex facts or actions cannot both be true (line 5). It is specified in Eq. 5.7.

$$\neg pf_i(m) \vee \neg pf_j(m), \neg pa_i(m) \vee \neg pa_j(m), \tag{5.7}$$

where $pf_i(m)$ and $pf_j(m)$ represent two mutex facts in fact level m, and $pa_i(m)$ and $pa_j(m)$ represent two mutex actions in action level m.

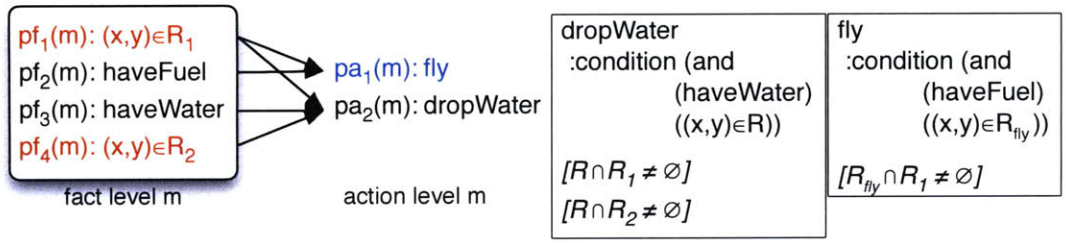


Figure 5-7: The conditions of action `dropWater` and `fly` are listed on the right. $(x, y) \in R_1$ and `haveFuel` are the resolved conditions of `fly`. $(x, y) \in R_1$, $(x, y) \in R_2$ and `haveWater` are the resolved conditions of `dropWater`.

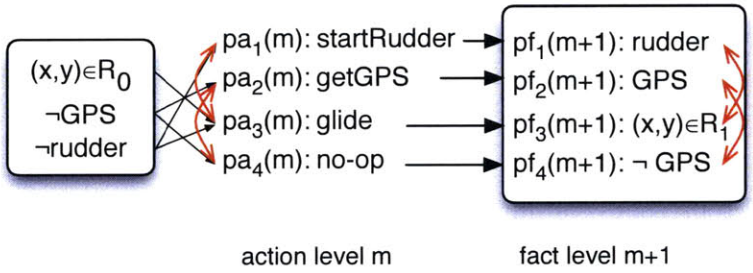


Figure 5-8: `startRudder` and `glide` are mutex actions. `getGPS` and no-op are mutex actions. `rudder` and $(x, y) \in R_1$ are mutex facts. `GPS` and `¬GPS` are mutex facts.

For example, in Fig. 5-8, the corresponding constraint encoding is $\neg pa_1(m) \vee \neg pa_3(m)$, $\neg pa_2(m) \vee \neg pa_4(m)$, $\neg pf_1(m+1) \vee \neg pf_3(m+1)$, and $\neg pf_2(m+1) \vee \neg pf_4(m+1)$.

6. If action *a-end* in action level j is true, then the time between *a-end* and its corresponding *a-start* action is within the duration bounds of the durative action a (Line 6).

$$pa(j) \implies d_{lb} \leq (j - i + 1) * \Delta t \leq d_{ub}, \tag{5.8}$$

where $pa(j)$ is the propositional variable representing an *-end* action in action level j, its matching *-start* action is in action level i, and $[d_{lb}, d_{ub}]$ are the lower and upper bounds on the duration of the corresponding durative action.

This encoding rule is specifically for $\mathrm{K_{DA}}$, to enforce the flexible temporal bounds on the duration of durative actions. It will be explained in Chapter 6.

7. If continuous region r in fact level m is true, then the state variables for fact

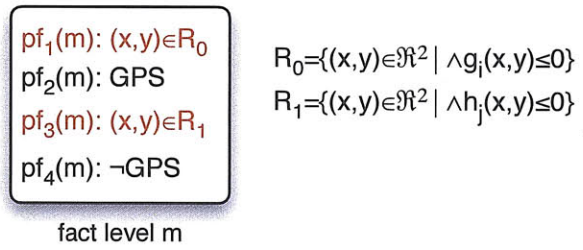


Figure 5-9: An example of fact level m contains continuous regions and literals. Continuous regions R_0 and R_1 are specified by conjunctions of linear inequalities.

level m satisfy r (Line 7). It is specified in Eq. 5.9.

$$pf(m) \implies \mathbf{x}(m) \in r, \tag{5.9}$$

where $pf(m)$ represents continuous region r in fact level m.

For example, in Fig. 5-9, the corresponding constraint encoding is $pf_1(m) \implies \wedge g_i(x,y) \leq 0$, and $pf_3(m) \implies \wedge h_j(x,y) \leq 0$.

8. If hybrid action a with specified dynamics in action level m is true, then (1) the control variables for action level m are within the actuation limits of action a, and (2) the state variables for fact level m and $m+1$ and the control variables for action level m satisfy a's state equation (Line 8). They are specified in Eq. 5.10.

$$pa(m) \implies \begin{cases} (1)\ \ \mathbf{u}(m) \in [\mathbf{u}_{lb}, \mathbf{u}_{ub}] \\ (2)\ \ \mathbf{x}(m+1) = A \times \mathbf{x}(m) + B \times \mathbf{u}(m) \end{cases} \tag{5.10}$$

where $pa(m)$ represents action a in action level m, $[\mathbf{u}_{lb}, \mathbf{u}_{ub}]$ is a's actuation limits, A and B are the matrices of constants in a's state equation.

For example, in Fig. 5-10, the corresponding constraint encoding is in Eq. 5.11.

$$pa_1(m) \implies \begin{cases} (1)\ \ vx(m) \in [vx_{lb}, vx_{ub}],\ vy(m) \in [vy_{lb}, vy_{ub}] \\ (2)\ \ x(m+1) = x(m) + vx(m) * d,\ y(m+1) = y(m) + vy(m) * d \end{cases} \tag{5.11}$$

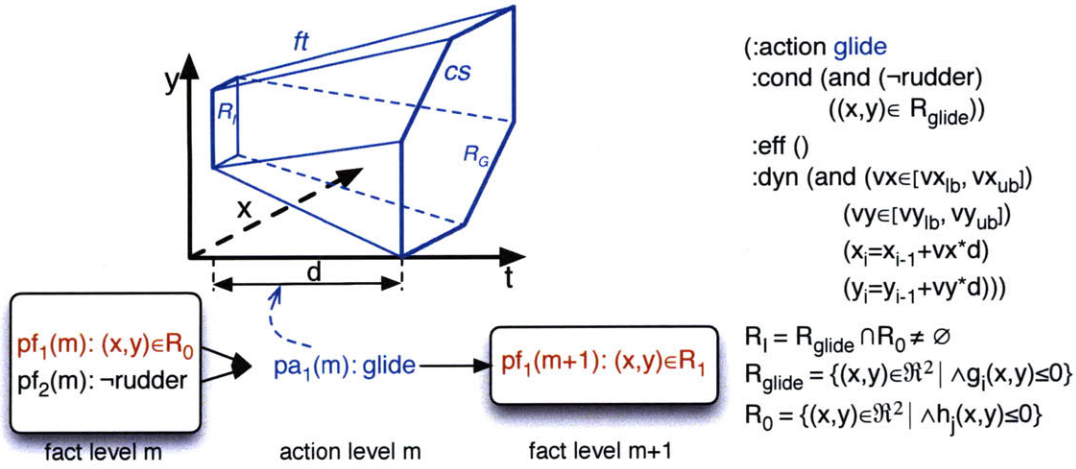


Figure 5-10: Action `glide` in action level m. Its type definition is listed on the right. R_{glide} is its continuous condition, specified by a conjunction of linear inequalities. R_0 is its resolved condition in fact level m, also specified by a conjunction of linear inequalities.

9. The state variables satisfy the external constraint set $\mathcal{C}$ for all fact levels (Line 9). It is specified in Eq. 5.12.

$$\forall m = 1, \ldots, k,\ \mathcal{C}(\mathbf{x}(m)) \tag{5.12}$$

where k is number of fact levels in the Hybrid Flow Graph.

For example, as in Fig. 5-11, the external constraint set is for obstacle avoidance and within map region. $\mathcal{C} = \{\vee g_i(\mathbf{x}) \leq 0,\ \vee h_j(\mathbf{x}) \leq 0,\ \wedge f_k(\mathbf{x}) \leq 0\}$. The corresponding constraint encoding is $\forall m = 1, \ldots, k,\ (\vee g_i(\mathbf{x}(m)) \leq 0) \wedge (\vee h_j(\mathbf{x}(m)) \leq 0) \wedge\ (\wedge f_k(\mathbf{x}(m)) \leq 0)$.

10. In each action level, either both action A_i-*start* and action A_j-*start* take place, or neither of them take place, where action A_i and action A_j are created for two goal episodes that share the same start event (Line 10):

$$\forall k = 1, \ldots, K\ (A_i\text{-}start(k) \wedge A_j\text{-}start(k))\ \vee\ (\neg A_i\text{-}start(k) \wedge \neg A_j\text{-}start(k)) \tag{5.13}$$

Similarly, when action A_i and action A_j are created for two goal episodes that

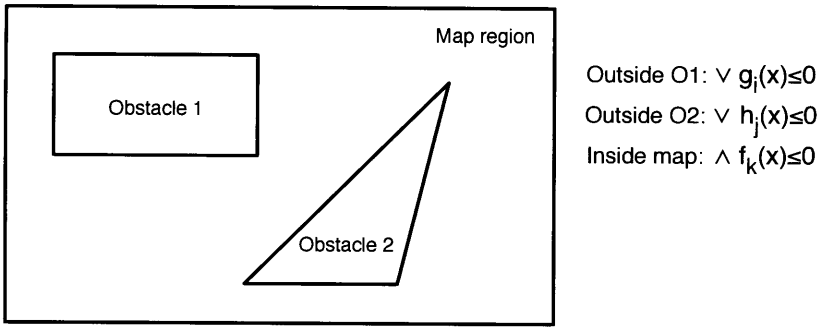


Figure 5-11: An example of external constraints: within the map region, and outside the obstacles.

share the same end event:

$$\forall k = 1, \ldots, K \ (A_i\text{-}end(k) \wedge A_j\text{-}end(k)) \ \vee \ (\neg A_i\text{-}end(k) \wedge \neg A_j\text{-}end(k)) \quad (5.14)$$

This encoding rule is specifically for K_{QSP}, to enforce the temporal relations of goal episodes in the input QSP. It will be explained in Chapter 7.

11. All simple temporal constraints in the input QSP are satisfied (Line 11):

$$\forall \text{ simple temporal constraint} \{\langle e_i, e_j \rangle, [lb, ub]\} \in \mathcal{QSP}, \ e_j - e_i \in [lb, ub]. \quad (5.15)$$

This encoding rule is specifically for K_{QSP}, to enforce the temporal constraints on events in the input QSP. It will be explained in Chapter 7.

Recall the review of Blackbox in Section 5.1, Rule 1-5 are similar to the constraint encoding rules in Blackbox. The continuous resolved conditions or conditions, which do not exist in Blackbox, are the major difference in Rule 1-5. Rule 6-11 are special rules for hybrid planning, and do not exist in the Blackbox framework.

In this chapter I reviewed Blackbox, and introduced the high-level algorithm in K_{AA}, in comparison with the high-level algorithm in Blackbox. I also described the constraint encoding of the Hybrid Flow Graph. In the next chapter, I will introduce K_{DA}, the Kongming planner that handles durative actions with flexible temporal bounds.

Alg. 10 EncodeMLQP&Solve(k-level hybrid flow graph: $hfg(k)$, state variables: $\mathbf{x}$, control variables: $\mathbf{u}$, propositional variables for facts: $\mathbf{pf}$, propositional variables for actions: $\mathbf{pa}$, initial conditions: $\mathcal{I}$, goal conditions: $\mathcal{G}$, hybrid action types: $\mathcal{HT}$, external constraint set: $\mathcal{C}$, objective function: f) **returns** k-level optimal plan: *solution*

1: Rule 1: All facts in fact level 1 are true
2: Rule 2: Facts that are true in fact level k contain $\mathcal{G}$
3: Rule 3: Fact in fact level m implies at least one of the actions in action level $m-1$ that cause the fact
4: Rule 4: Action in action level m implies all resolved conditions in fact level m
5: Rule 5: Mutex facts or actions cannot both be true
6: Rule 6: Action *a-end* in action level j implies the time between *a-end* and its corresponding *a-start* action is within the duration bounds of a
7: Rule 7: Continuous region r in fact level m implies $\mathbf{x}(m) \in r$
8: Rule 8: Action a with specified dynamics in action level m implies (1) $\mathbf{u}(m)$ are within the actuation limits of action a, and (2) $\mathbf{x}(m)$, $\mathbf{x}(m+1)$, $\mathbf{u}(m)$ satisfy a's state equation
9: Rule 9: $\mathbf{x}$ satisfy $\mathcal{C}$ for all fact levels
10: Rule 10: Two actions start in the same action level, if they represent two goal episodes that share the same start event; Two actions end in the same action level, if they represent two goal episodes that share the same end event
11: Rule 11: All simple temporal constraints in the input QSP are satisfied
12: **if** solution to MLQP found **then**
13: **return** *solution*
14: **else**
15: **return** *nil*
16: **end if**

Chapter 6

K_{DA}: Planning with Durative Actions

Contents

Chapter 4 and 5 introduced K_{AA}, the core planner of Kongming that plans for a final goal state, with hybrid atomic actions. Recall that Kongming is divided into three planners. K_{DA} plans with hybrid durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. K_{QSP} plans for QSPs instead of a final goal state, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. This chapter introduces K_{DA}, and next chapter will introduce K_{QSP}.

The core of K_{DA} is a reformulation scheme that converts hybrid durative actions

to hybrid atomic actions. The reformulation builds upon the encoding in LPGP [LF02]. LPGP encodes each discrete durative action as a *start*, an *end* and one or more intermediate actions. K_{DA} combines the LPGP encoding with flow tubes to reformulate hybrid durative actions as hybrid atomic actions.

In this chapter, I first review LPGP, and then introduce the reformulation of hybrid durative actions.

6.1 LPGP

The reformulation of hybrid durative actions as hybrid atomic actions builds upon the encoding in LPGP [LF02]. In this section, I review the LPGP encoding.

LPGP encodes each discrete durative action A as a *start* action, A-start, an *end* action, A-end, and one or more invariant checking actions, A-inv. A-start and A-end encode the start and the end of A, while A-inv encodes the period when A is maintained. Two propositions, A-ing-inv and i-A-ing-inv, are introduced as flags, in order to sequence the these actions together. The net effect is to ensure that A-start is performed first, followed by one or more A-inv, and ending with A-end.

As shown in Fig. 6-1, between a *start* action and an *end* action, there are one or more actions for invariant checking in the middle of a durative action.

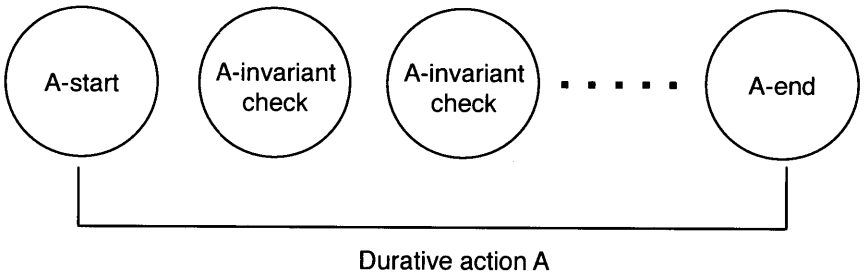


Figure 6-1: A durative action is formulated as a *start* action at the start, an *end* action at the end, and a series of actions for invariant checking in the middle.[LF02]

I review the encoding of the *start* action, the *inv* (invariant checking) action, and the *end* action as follows, in accordance with Fig. 6-2. On the left-hand side of the figure, there is the specification of a durative action A. On the right-hand side of the

figure, there is the specification of the *start* action, the *inv* action, and the *end* action.

- The conditions of `A-start`: the `start` conditions of `A`.
- The effects of `A-start`: the `start` effects of `A` and literal `A-ing-inv`.
- The conditions of `A-inv`: the `overall` conditions of `A` and literal `A-ing-inv`
- The effects of `A-inv`: the `overall` effects of `A` and literals `A-ing-inv` and `i-A-ing-inv`.
- The conditions of `A-end`: the `end` conditions of `A` and literal `i-A-ing-inv`.
- The effects of `A-end`: the `end` effects of `A` and literals ¬`A-ing-inv` and ¬`i-A-ing-inv`.
- LPGP stores the duration value of each action in a separate file, while adding a special duration field to each of the start and end actions.

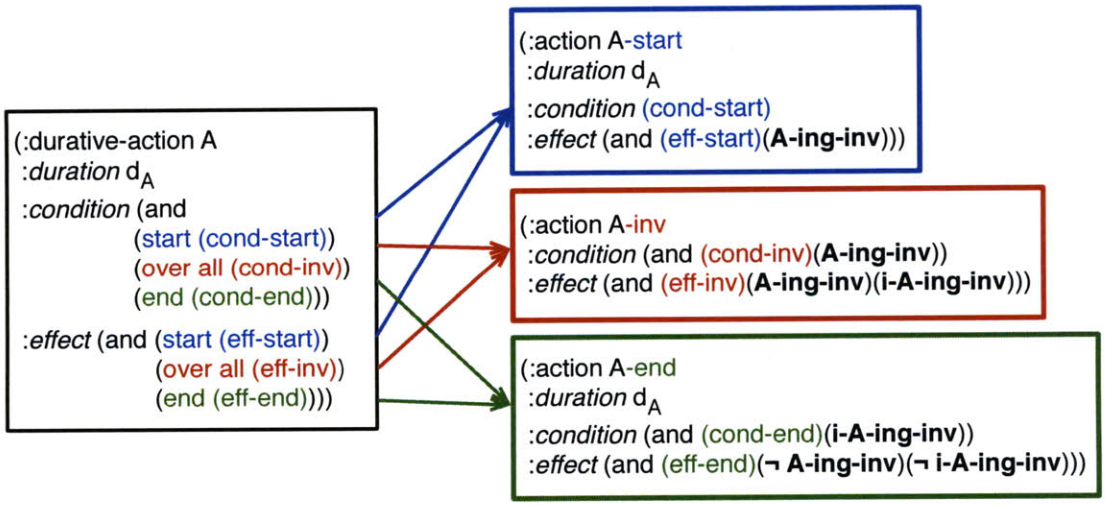


Figure 6-2: The LPGP encoding of a durative action. On the left-hand side of the figure, there is the specification of a durative action A. On the right-hand side of the figure, there is the specification of the *start* action, the *inv* action, and the *end* action.

LPGP applies this encoding of durative actions to Graphplan [BF97], with a few modifications to Graphplan. I review the modifications as follows.

- LPGP has a stronger requirement than Graphplan to ensure that actions do not interfere with one another. It requires that two actions to be mutex if they share the same *end* effect. This is to avoid situations where two actions have to be synchronized to end at exactly the same time.

- No no-op actions are constructed for the literals that have an *-inv* suffix. This forces the invariant checking actions to be used to propagate these facts between levels, ensuring that the invariant conditions are checked as the propagation is carried out.

- The graph search in Graphplan is modified, so that constraints on action durations can be incorporated. When an *end* action is selected, a temporal constraint is added. The temporal constraint requires that the total duration between the *start* and *end* actions of the durative action must equal the duration of the durative action.

6.2 K_{DA}

Thus far I have reviewed the LPGP approach for encoding discrete durative actions. In this section I introduce K_{DA}, by starting with the input and output.

6.2.1 Input & Output

Recall that the problem statement for K_{DA} was defined in Section 3.2. I review them here.

The input consists of initial conditions $\mathcal{I}$ (Def. 4), final goal conditions $\mathcal{G}$ (Def. 20), hybrid durative action types $\mathcal{DA}$ (Def. 10), external constraints $\mathcal{C}$ (Def. 16), and an objective function f (Def. 17). The output consists of an optimal hybrid plan $\mathcal{P}_{DA}$ (Def. 21).

As shown in Fig. 6-3, there are three components in K_{DA}. The first component is Reformulation. It encodes hybrid durative action types (Def. 10) into hybrid atomic action types (Def. 23). The second component is K_{AA}. It is in charge of the

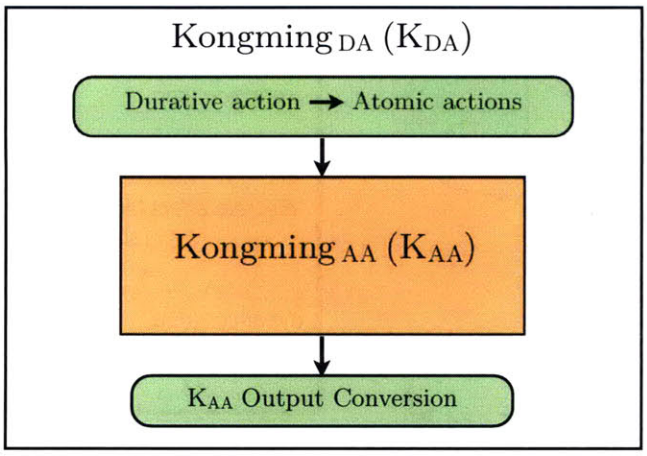


Figure 6-3: K_{DA} overview diagram. There are three components: reformulation of hybrid durative actions types into hybrid atomic action types, K_{AA}, and K_{AA} output conversion.

core planning and was introduced in Chapter 4 and 5. The third component is the K_{AA} output conversion. It converts the output of K_{AA} to the output of K_{DA}. In the following sections, I first describe the Reformulation part and then the Output Conversion part.

6.2.2 Reformulation

K_{DA} generalizes the LPGP encoding for hybrid durative actions. Because the LPGP encoding only applies to actions with discrete conditions and effects, the main challenge is how to incorporate the flow tubes of the hybrid actions in the encoding.

Fig. 6-4 shows a simple flow tube representation of a hybrid durative action with flexible duration. The specification of a hybrid durative action is listed on the right. Its flow tube representation is on the left. The *start*, *overall* and *end* conditions need to be checked at the start, in the middle and at the end of the flow tube. The *start*, *overall* and *end* discrete effects need to be added at the start, in the middle and at the end of the flow tube.

K_{DA} encodes each hybrid durative action as a set of atomic actions, by "slicing" the flow tube into multiple pieces. All flow tube slices have the same duration, Δt. K_{DA} combines the flow tube slices with the LPGP encoding, to ensure that first, the

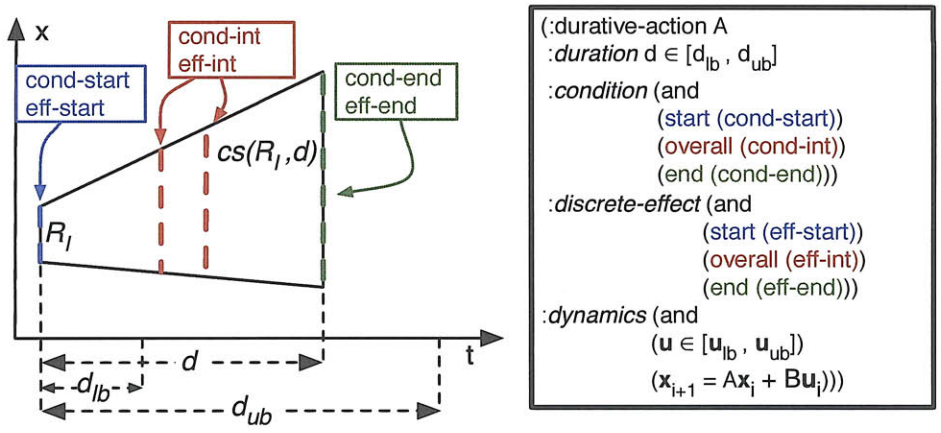


Figure 6-4: The specification of a hybrid durative action is listed on the right. Its flow tube representation is on the left. *cond-start* represents the set of *start* conditions; *cond-int* (*intermediate*) represents the set of *overall* conditions; *cond-end* represents the set of *end* conditions. *eff-start* represents the set of *start* discrete effects; *eff-int* represents the set of *overall* discrete effects; *eff-end* represents the set of *end* discrete effects. The duration of the action is flexible. The *start*, *overall* and *end* conditions need to be checked at the start, in the middle and at the end of the flow tube. The *start*, *overall* and *end* discrete effects need to be added at the start, in the middle and at the end of the flow tube.

conditions at various stages are enforced at the beginning of each flow tube slice; second, the effects at various stages are added at the end of each flow tube slice; and third, the *start* flow tube slice is performed first, followed by one or more *intermediate* flow tube slices, and ending with the *end* flow tube slice.

As shown on the left-hand side of Fig. 6-5, first, the flow tube in Fig. 6-4 is divided into multiple flow tube slices, each having duration Δt. Second, as shown on the right-hand side of Fig. 6-5, the flow tube slices are combined with the LPGP encoding. More specifically, the first flow tube slice corresponds to atomic action *A-start*. The last flow tube slice corresponds to atomic action *A-end*. The flow tube slices in the middle correspond to atomic action *A-int*. The duration for all the atomic actions is Δt. Moreover, the atomic actions all keep the same dynamics as in the durative action.

This encoding of a hybrid durative action not only discretizes the flow tube into slices with length of Δt each, but also uses the conditions of the action type at different stages to prune the invalid regions in state space and hence invalid trajectories in

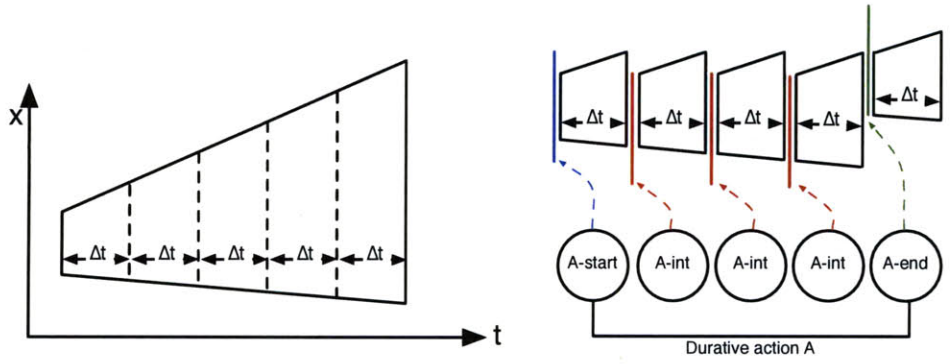


Figure 6-5: K_{DA} reformulates a hybrid durative action into hybrid atomic actions by combining the flow tube slices with the LPGP encoding. On the left-hand side, the flow tube of a hybrid durative action is divided into slices, each with length Δt. On the right-hand side, the flow tube slices are combined with the LPGP encoding. The blue line represents the continuous condition of *A-start*. The red lines represent of the continuous condition of *A-int*. The green line represents the continuous condition of *A-end*. The initial region of a flow tube slice is the intersection of the continuous condition of its corresponding atomic action and the resolved goal region of its previous flow tube slice.

the flow tube. As shown on the right-hand side of Fig. 6-5, the initial region of each flow tube slice is constrained by the condition of its corresponding atomic action. As described in Section 4.2.2 and Section 4.2.3, the initial region of a flow tube is the intersection of a resolved goal region and the continuous condition of the hybrid atomic action, where a resolved goal region is the intersection of the goal region of the previous flow tube and unit-clause external constraints. In other words, initial region = resolved goal region $\bigcap$ continuous condition. Therefore, the initial region of the following flow tube is a subset of the resolved goal region of the previous flow tube.

The procedure of reformulating a hybrid durative action type is given in Alg. 11. The input of the reformulation is a hybrid durative action type, as reviewed in Section 7.1. The output of the reformulation is a set of hybrid atomic action types that satisfy the definition reviewed in Section 6.2.1. It builds upon the LPGP encoding reviewed in Section 6.1. I describe the similarities and differences between my encoding and LPGP as follows, in accordance with the pseudo code in Alg. 11.

Similarities: K_{DA} encodes a hybrid durative action A into three hybrid atomic actions: A-start, A-int (for *intermediate*), and A-end. A-start and A-end encode the

start and the end of A, while A-int encodes the intermediate period after A starts and before A ends. Two propositions, A-ing-int and i-A-ing-int, are introduced as flags, in order to sequence the three actions together. The net effect is to ensure that A-start is performed first, followed by one or more A-int, and ending with A-end. See details of the similarities as follows.

- Three atomic action types are created for each durative action type A: *A-start*, *A-int*, and *A-end* (line 1, 6, 11 in Alg. 11).
- The conditions of *A-start* are the *start* conditions of A (line 3).
- The discrete effects of *A-start* are the *start* discrete effects of A and literal *A-ing-int* (line 4).
- The conditions of *A-int* are the *overall* conditions of A and literal *A-ing-int* (line 8).
- The discrete effects of *A-int* are the *overall* discrete effects of A and literals *A-ing-int* and *i-A-ing-int* (line 9).
- The conditions of *A-end* are the *end* conditions of A and literal *i-A-ing-int* (line 13).
- The discrete effects of *A-end* are the *end* discrete effects of A and literals $\neg$*A-ing-int* and $\neg$*i-A-ing-int* (line 14).

Differences: K_{DA} divides the flow tube of a hybrid durative action A into smaller flow tubes, each of which has a fixed duration Δt.

- Each flow tube represents a hybrid atomic action, which can be *A-start*, *A-int*, or *A-end*.
- The duration of each atomic action is Δt (line 2, 7, 12 in Alg. 11).
- Each atomic action keeps the same dynamics as in durative action A in order to construct each flow tube (line 5, 10, 15).

Alg. 11 ConvertHybridAction(hybrid durative action type: A) **returns** a set of hybrid atomic action types: $aSet$. {A.cond-start: the set of *start* conditions of A; A.cond-int: the set of *overall* conditions of A; A.cond-end: the set of *end* conditions of A; A.eff-start: the set of *start* discrete effects of A; A.eff-int: the set of *overall* discrete effects of A; A.eff-end: the set of *end* discrete effects of A.}

1: create atomic action type A-*start*
2: A-*start*.duration $\leftarrow \Delta t$
3: A-*start*.condition $\leftarrow A$.cond-start
4: A-*start*.discrete-effect $\leftarrow$ (A.eff-start & A-*ing-int*)
5: A-*start*.dynamics $\leftarrow A$.dynamics
6: create atomic action type A-*int*
7: A-*int*.duration $\leftarrow \Delta t$
8: A-*int*.condition $\leftarrow$ (A.cond-int & A-*ing-int*)
9: A-*int*.discrete-effect $\leftarrow$ (A.eff-int & A-*ing-int* & *i-A-ing-int*)
10: A-*int*.dynamics $\leftarrow A$.dynamics
11: create action type A-*end*
12: A-*end*.duration $\leftarrow \Delta t$
13: A-*end*.condition $\leftarrow$ (A.cond-end & *i-A-ing-int*)
14: A-*end*.discrete-effect $\leftarrow$ (A.eff-end & $\neg A$-*ing-int* & $\neg$*i-A-ing-int*)
15: A-*end*.dynamics $\leftarrow A$.dynamics
16: **return** $aSet$.add(A-*start*, A-*inv*, A-*end*)

Now I discuss what the appropriate value of Δt can be.

Recall from Chapter 4 that all actions in the input to K_{AA} are hybrid atomic actions, which all have fixed and equal duration. Hence, the duration of each action level in a Hybrid Flow Graph is fixed and equal. Since the hybrid atomic action types that are output from the reformulation have duration Δt, the duration of each action level in a Hybrid Flow Graph is Δt.

Recall from the definition of the dynamic state equation of a hybrid action type (Def. 14), that time is discretized as $\langle t_0, t_1, \ldots \rangle \in \mathcal{R}^{\mathcal{N}}$. One option is to set the duration of each hybrid atomic action (Δt) to the time increment in the discretization, $t_i - t_{i-1}$. This ensures that the flow tube computation for each hybrid atomic action is as accurate as the state equation. For some dynamic systems, where the time increment in the state equation is not very small, this option works well. For example, the time increment for an autonomous underwater vehicle can be 2 seconds [ody].

However, for other more agile dynamic systems, the time increment is small. For example, the time increment for an unmanned air vehicle [Léa05] or a bi-ped system

[Hof06] can be 0.1 seconds. Because Δt is also the length of each action level in a Hybrid Flow Graph, if Δt is set this small, for a planning problem of horizon on the order of hours, the number of levels in the Hybrid Flow Graph will be on the order of 10^4. In this case, constructing the Hybrid Flow Graph and searching for a plan become too computationally expensive. Thus, in this case, K_{DA} sets Δt bigger than the time increment of the dynamic system under control, but as small as computation power allows. I discuss the value of Δt further in the empirical results in Chapter 8.

6.2.3 K_{DA}-specific Designs in K_{AA}

Recall from the K_{AA} planner, introduced in Chapter 4 and 5, that a few features in K_{AA} are created specifically for K_{DA} and were not explained in detail in Chapter 4 and 5. I explain them in detail in this section.

The first feature is that no no-op actions are constructed for the facts that have an "-int" suffix. This feature is similar to the second modification LPGP makes to Graphplan in order to apply its encoding of durative actions to Graphplan (Section 6.1). This feature enforces that only the intermediate (-int) actions can be used to propagate the "-int" facts between levels.

The second feature is to enforce that the sum of all Δt between the *-start* and *-end* atomic actions of the same durative action is within the duration bounds of the durative action. This simple temporal constraint is enforced in the MLQP constraint encoding of K_{AA}, if the *-end* action takes place. It is specified by the following formula.

$$pa(j) \implies d_{lb} \leq (j - i + 1) * \Delta t \leq d_{ub}, \tag{6.1}$$

where $pa(j)$ is the propositional variable representing an *-end* action in action level j, its matching *-start* action is in action level i, and $[d_{lb}, d_{ub}]$ are the lower and upper bounds on the duration of the corresponding durative action.

This feature is similar to the third modification LPGP makes to Graphplan, as reviewed in Section 6.1. The purpose of both is to enforce the temporal constraint on action duration. In LPGP, the modification happens in the back-chaining search

in Graphplan. When an *end* action is selected, LPGP adds to the linear program a temporal constraint that requires that the total duration between the *start* and *end* actions of the durative action must equal the duration of the durative action. In K_{AA}, the feature takes place in the MLQP constraint encoding, described in Chapter 5. The main difference is that because in K_{DA} the durative actions have flexible durations, K_{AA} adds to the MLQP encoding a temporal constraint that requires that the total duration should be within the duration bounds of the durative action.

As shown in Fig. 6-6, for example, action *A-end* is in action level $i+2$, its matching *A-start* action is in action level i. If *A-end* takes place, then the time between the beginning of action level i and the end of action level $i+2$, $3 * \Delta t$, needs to be within the duration bounds on A's duration.

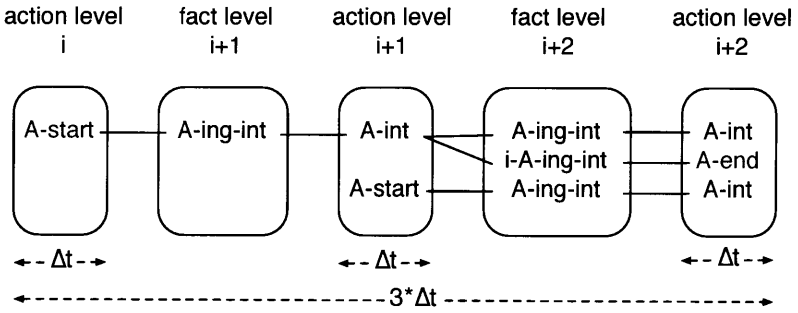


Figure 6-6: Action *A-end* is in action level $i+2$, its matching *A-start* action is in action level i. If *A-end* takes place, then the time between the beginning of action level i and the end of action level $i+2$, $3 * \Delta t$, needs to be within the duration bounds on A's duration.

6.2.4 Reformulation Correctness

In this section, I show that the reformulation is correct. In other words, the output of the reformulation is equivalent to the input of the reformulation. Recall that a hybrid durative action type consists of a duration, conditions, discrete effects, and dynamics. I show that each component is kept equivalent in the hybrid atomic action types reformulated from the durative action type.

The duration of a hybrid durative action type is flexible and bounded by a lower and upper bound, $d \in [d_{lb}, d_{ub}]$. As described in Section 6.2.3, a K_{DA}-specific feature in

K_{AA} enforces the sum of all Δt between the *-start* and *-end* atomic actions of the same durative action to be within the duration bounds of the durative action. Therefore, the duration bounds of the durative action are satisfied by the total duration of the sequence of atomic actions reformulated from the durative action.

The conditions of a hybrid durative action type are specified in three stages: *start*, *overall*, and *end*. As described in Alg. 11, the conditions in the *start* stage are included in the conditions of the *-start* atomic action. Likewise for the conditions in the *overall* and the *end* stage.

The discrete effects of a hybrid durative action type are specified in three stages: *start*, *overall*, and *end*. As described in Alg. 11, the discrete effects in the *start* stage are included in the discrete effects of the *-start* atomic action. Likewise for the discrete effects in the *overall* and the *end* stage.

The dynamics of a hybrid durative action type are kept the same in all the corresponding hybrid atomic action types.

6.2.5 Output Conversion

Recall from Chapter 3 that the optimal hybrid plan that is output from K_{DA} is slightly different from the output plan from K_{AA}. I present the conversion from K_{AA}'s output to K_{DA}'s output in this section.

The main difference between the two output plans is that, in K_{AA}'s output, the actions in the optimal action sequence are atomic, whereas in K_{DA}'s output, they are durative.

More specifically, as defined in Chapter 3, the optimal action sequence output from K_{AA} is in the form of $\mathbf{A}^*_{AA} = \{\mathbf{a}^*(t_1), \mathbf{a}^*(t_2), \ldots, \mathbf{a}^*(t_{N-1})\}$, where $\forall t_i = t_1, \ldots, t_{N-1}, t_{i+1} - t_i$ is equal to a discretization constant Δt, $\mathbf{a}^*(t_i)$ is the set of actions to be performed at time t_i, and the actions are instantiated from the hybrid atomic action types $\mathcal{AA}$. For example, in the underwater domain, the optimal action sequence is as follows. At time t_1, {`glide-start`}; at time t_2, {`glide-int`, `setRudder-start`}; at time t_3, {`glide-int`, `setRudder-int`}; at time t_4, {`glide-end`, `setRudder-end`}; at time t_5, {`descend-start`, `setGulper-start`}; at time t_6, {`descend-int`, `setGulper-int`};

at time t_7, {`descend-end`, `setGulper-end`}; at time t_8, {`takeSample-start`}; at time t_9, {`takeSample-int`}; at time t_{10}, {`takeSample-int`}; at time t_{11}, {`takeSample-end`}.

On the other hand, as defined in Chapter 3, the optimal action sequence output from $\mathrm{K_{DA}}$ is in the form of $\mathbf{A^*}_{DA} = \{\mathbf{a^*_d}(t_{n_1}), \mathbf{a^*_d}(t_{n_2}), \ldots, \mathbf{a^*_d}(t_{n_m})\}$, where $\{t_{n_1}, \ldots, t_{n_m}\} \subseteq \{t_1, \ldots, t_{N-1}\}$, $\mathbf{a^*_d}(t_{n_i})$ is the set of $\langle a, d\rangle$ pairs at time t_{n_i}; in each pair, a is an action instantiated from the hybrid durative action types $\mathcal{DA}$ that starts at time t_{n_i}, and d is its duration. For example, corresponding to the optimal action sequence example introduced previously for $\mathrm{K_{AA}}$, if the discretization constant $\Delta t = 1$, then the optimal action sequence output from $\mathrm{K_{DA}}$ is as follows. At time t_1, {$\langle$`glide`, 4$\rangle$}; at time t_2, {$\langle$`setRudder`, 3$\rangle$}; at time t_5, {$\langle$`descend`, 3$\rangle$, $\langle$`setGulper`, 3$\rangle$}; at time t_8, {$\langle$`takeSample`, 4$\rangle$}.

The output conversion module in $\mathrm{K_{DA}}$ is responsible for converting the optimal action sequence of $\mathrm{K_{AA}}$, named $\mathbf{A^*}_{AA}$, to the optimal action sequence of $\mathrm{K_{DA}}$, named $\mathbf{A^*}_{DA}$. Output Conversion involves two steps. First, it identifies the *-start* actions in $\mathbf{A^*}_{AA}$. Second, it counts how many time steps there are between each pair of *A-start* and *A-end* actions in $\mathbf{A^*}_{AA}$. If there are n time steps, then the duration of action A is $n \times \Delta t$. The pseudo code of Output Conversion in $\mathrm{K_{DA}}$ is shown in Alg. 12.

Alg. 12 OutputConversion$_{\mathrm{DA}}$ (optimal action sequence of $\mathrm{K_{AA}}$: $\mathbf{A^*}_{AA}$) **returns** optimal action sequence of $\mathrm{K_{DA}}$: $\mathbf{A^*}_{DA}$

```
create hash table A*_DA {Stores time points as keys and action-duration pairs as values}
for t_i = t_1, ..., t_{N-1} do
  for each action a in a*(t_i) do
    if a contains -start then
      a ← (-start removed from a) {Identifies the -start actions}
      t_j ← t_{i+1}
      while a-end is not in a*(t_j) do
        j ← j + 1 {Looks for the corresponding -end action}
      end while
      d ← t_j − t_i {Computes action duration}
      add (t_i, ⟨a, d⟩) to A*_DA
    end if
  end for
end for
return A*_DA
```

This chapter presented K_{DA}, the planner that plans with hybrid durative actions with flexible durations, by reformulating them to atomic actions. In the next chapter I will introduce K_{QSP}.

Chapter 7

K_{QSP}: Planning for A Qualitative State Plan

Contents

In Chapter 6, I introduced K_{DA}. It plans with hybrid durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. K_{QSP} plans for QSPs instead of a final goal state, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. This chapter introduces K_{QSP}.

The core of K_{QSP} is a reformulation scheme, which reformulates a QSP to durative actions and a final goal state. The reformulation is based on the idea that each goal state of the QSP can be achieved through the conditions of an action. K_{QSP} creates a hybrid durative action for each goal episode of the QSP. The conditions of the hybrid durative action include all the state constraints of the goal episode. In order to make the effects of the action true, the conditions have to be satisfied, which enforces the state constraints to be achieved.

7.1 Input & Output

Recall that the problem statement for K_{QSP} was defined in Section 3.1. I review them here.

The input consists of initial conditions $\mathcal{I}$ (Def. 4), a QSP $\mathcal{QSP}$ (Def. 5), hybrid durative action types $\mathcal{DA}$ (Def. 10), external constraints $\mathcal{C}$ (Def. 16), and an objective function f (Def. 17). The output consists of a hybrid optimal plan $\mathcal{P}_{QSP}$ (Def. 18).

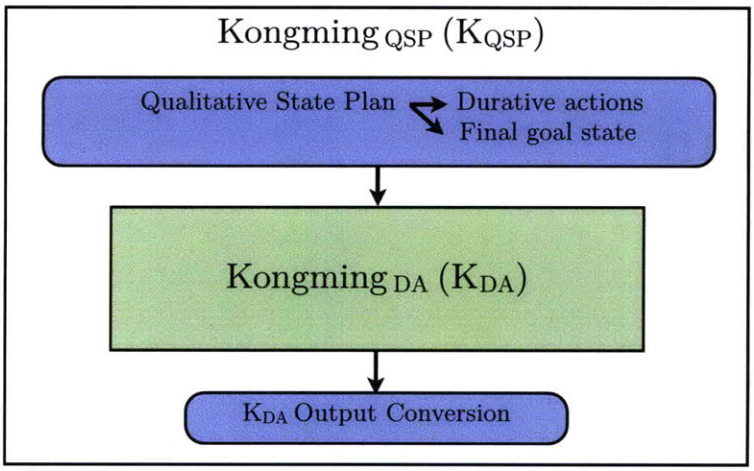


Figure 7-1: K_{QSP} overview diagram. There are three components: reformulation of a qualitative state plan (QSP), K_{DA}, and K_{DA} output conversion.

As shown in Fig. 7-1, there are three components in K_{QSP}. The first component is Reformulation. It encodes a qualitative state plan (QSP) into a set of hybrid durative action types (Def. 10) and a final goal state (Def. 20). The second component is K_{DA}, introduced in Chapter 6. The third component is the K_{DA} output conversion. It converts the output of K_{DA} to the output of K_{QSP}. In the following sections, I first describe the Reformulation part and then the Output Conversion part.

7.2 Reformulation

It is important to be able to plan for multiple goals at different times of a task. Often we would like to specify an ordering of goal achievement. For example, delivery services should deliver all priority packages first and then deliver the regular mail. Every so

often we also want to specify goals to achieve in parallel. For example, in an unmanned aerial vehicle fire-fighting mission, a reconnaissance vehicle should monitor the fire situation before, during and after other vehicles put out the fire. Such temporally extended goals specify goal requirements at various times of the execution of a plan. In this thesis, temporally extended goals are represented as qualitative state plans (QSPs), as defined in Section 3.1.3.

In this section, I introduce how to reformulate a QSP as a set of hybrid durative action types and a final goal state.

K_{QSP} reformulates a QSP into hybrid durative action types and a final goal state, by creating a hybrid durative action type for each goal episode, and encoding the fact that all goal episodes in the QSP are achieved as a final goal state. A QSP (Def. 5) was defined in Chapter 3. I first review the definition of a QSP.

A QSP consists of a set of events, each representing a point in time, a set of goal episodes, and a set simple temporal constraints on the events. A goal episode consists of a start event and an end event of the goal episode, as well as a set of state constraints that must hold at the start, for the duration, and at the end of the goal episode. Each state constraint is a conjunction of a linear system over the continuous state variables and a set of literals over the discrete state variables. Each simple temporal constraint in a QSP consists of a start event, an end event, and a lower and an upper bound on the time between the two events. An example of a QSP is shown in Fig. 7-2.

We observe that goal episodes in a QSP are similar to hybrid durative actions in that they are durative, and have constraints specified at the start, over the duration, and at the end. An encoding that represents goal episodes as durative actions may be used. The main difference is that goal episodes establish conditions to be satisfied, while actions establish effects to be produced. In addition, a QSP specifies which goal episodes precede and follow other goal episodes, while there is no specific description on how durative action types are ordered.

Fig. 7-3 shows the reformulation of one of the goal episodes in a QSP. On the left-hand side of the figure, there is a QSP of three goal episodes, as introduced in

Fig. 7-2. Let us focus on goal episode ge_2. On the right-hand side of the figure, the action created for goal episode ge_2 is listed. The duration of the action is bounded by the simple temporal constraint on the start and end events of the goal episode, if there is any. The conditions of the action include the state constraints of the goal episode, and the fact that all predecessors of the goal episode have been achieved. The discrete effect of the action at the end is the fact that this goal episode has been achieved. Dynamics are unspecified.

I explain the reformulation procedure as follows, in accordance with the pseudo code in Alg. 13.

- A hybrid durative action type a is created for every goal episode ge in the QSP (line 2).

- If there exists a simple temporal constraint in the QSP on the time between the start event and the end event of the goal episode (line 3), then the duration of a is bounded by the lower and upper bounds in the temporal constraint (line 4). Otherwise, the duration of a is unbounded, specified as $[0, +\infty)$ (line 6).

- The state constraints of ge are encoded as the conditions of a. More specifically, the *start* constraints of ge are encoded as the *start* conditions of a (line 8); the *overall* constraints of ge are encoded as the *overall* conditions of a (line 9); and the *end* constraints of ge are encoded as the *end* conditions of a (line 10).

- Another *start* condition of a requires that all the predecessors of ge have been achieved (line 8). Naturally, a goal episode cannot be started if any of its predecessors have not completed.

- The only discrete effect of a is that ge has completed, specified by a literal *ge-ended* (line 13). This effect is the *end* effect, because naturally, before the end of a goal episode, it cannot be considered completed. The *start* and *overall* effects of a are empty (line 11-12).

- The dynamics of a are unspecified (line 14), because the actions are "pseudo"

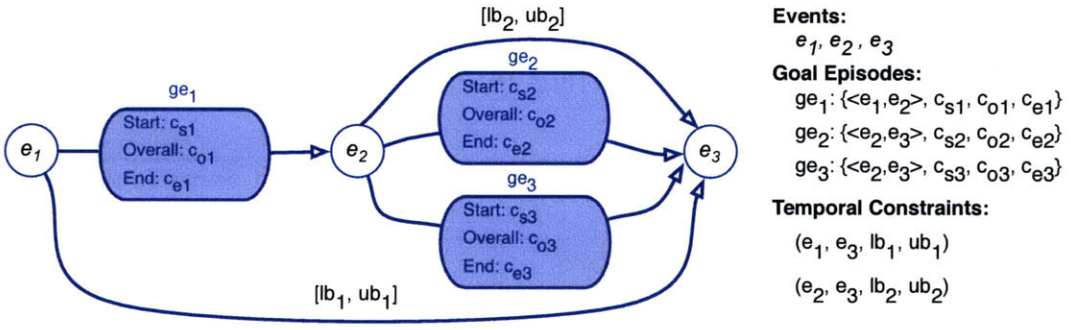


Figure 7-2: The events in the QSP are e_1, e_2 and e_3. The goal episodes are ge_1, ge_2 and ge_3. c_{s1}, c_{o1} and c_{e1} are the state constraints of ge_1. c_{s2}, c_{o2} and c_{e2} are the state constraints of ge_2. c_{s3}, c_{o3} and c_{e3} are the state constraints of ge_3. The temporal constraint on e_1 and e_3 has lower bound lb_1 and upper bound ub_1. The temporal constraint on e_2 and e_3 has lower bound lb_2 and upper bound ub_2.

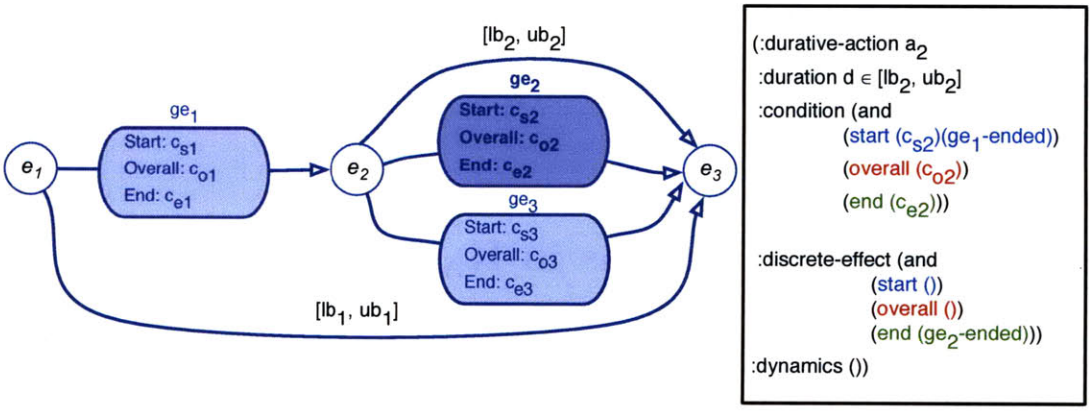


Figure 7-3: Hybrid durative action type a_2 is created for goal episode ge_2. ge_1-ended and ge_2-ended are literals, representing respectively the fact that ge_1 and ge_2 are achieved. c_{s2}, c_{o2} and c_{e2} are the constraints at different times of ge_2.

actions that do not produce any actual effects. They are created to enforce the state constraints of goals in the QSP.

- The final goal state fg requires that all goal episodes in the QSP are achieved, specified by $\wedge_i$ ge_i-*ended*, $\forall ge_i \in qsp$ (line 16).

Alg. 13 ReformulateQSP(a QSP: qsp) **returns** (a set of hybrid durative action types $aSet$, a final goal fg)

1: **for** each goal episode ge in qsp **do**
2: create hybrid durative action type a
3: **if** (ge.start-event, ge.end-event) $\in$ temporal constraints of qsp **then**
4: a.duration $\leftarrow [lb, ub]$
5: **else**
6: a.duration $\leftarrow [0, +\infty)$
7: **end if**
8: a.condition(start) $\leftarrow$ (ge.constraints(start) & $\wedge_i$ ge_i-*ended*) {Every ge_i is a predecessor of ge.}
9: a.condition(overall) $\leftarrow$ ge.constraints(overall)
10: a.condition(end) $\leftarrow$ ge.constraints(end)
11: a.discrete-effect(start) $\leftarrow \emptyset$
12: a.discrete-effect(overall) $\leftarrow \emptyset$
13: a.discrete-effect(end) $\leftarrow$ *ge-ended*
14: a.dynamics $\leftarrow \emptyset$
15: $aSet$.add(a)
16: fg.add(*ge-ended*)
17: **end for**
18: **return** ($aSet, fg$)

7.3 K_{QSP}-specific Designs in K_{AA}

Recall from the K_{AA} planner, introduced in Chapter 4 and 5, that a few features in K_{AA} are created specifically for K_{QSP} to incorporate the temporal information in a QSP, and were not explained in detail in Chapter 4 and 5. I explain them in detail in this section.

Recall that a QSP consists of three components: events, goal episodes, and temporal constraints. There are two types of temporal information associated with events. First, a goal episode is the predecessor of another goal episode, if the end event of the first

goal episode is the start event of the second goal episode. This temporal information is included in the reformulation introduced in the previous section. However, the second type of temporal information associated with events is ignored in the reformulation. Namely, by sharing the same start or end event, goal episodes are required to start or end at the same time. For example, in Fig. 7-2, ge_2 and ge_3 are required to start and end at the same time.

In addition, some temporal constraints are also ignored in the reformulation, namely, the temporal constraints whose start and end events do not belong to any goal episodes in the QSP. For example, in Fig. 7-2, the temporal constraint over e_1 and e_3 is not captured in the reformulation.

In summary, the K_{QSP}-specific features in K_{AA} consist of two types of temporal constraints:

1. If an event is the start (or end) event of multiple goal episodes, then these goal episodes should start (or end) at the same time. For example, ge_2 and ge_3 in Fig. 7-2.

2. Simple temporal constraints that are not over any specific goal episode in the QSP. For example, $[lb_1, ub_1]$ between e_1 and e_3 in Fig. 7-2.

As introduced in Chapter 5, Temporal Constraint 1 is included in the MLQP constraint encoding of K_{AA}. Suppose goal episode ge_i and goal episode ge_j share the same start event e, and suppose ge_i is encoded as hybrid durative action type A_i, and ge_j is encoded as hybrid durative action type A_j. Suppose there are K levels in the Hybrid Flow Graph. Then a constraint needs to be added to require that in any action level, either both action A_i-*start* and action A_j-*start* take place, or neither of them take place. It is specified as follows:

- $\forall k = 1, \ldots, K \ (A_i\text{-}start(k) \wedge A_j\text{-}start(k)) \ \vee \ (\neg A_i\text{-}start(k) \wedge \neg A_j\text{-}start(k))$

Likewise for two goal episodes sharing the same end event:

- $\forall k = 1, \ldots, K \ (A_i\text{-}end(k) \wedge A_j\text{-}end(k)) \ \vee \ (\neg A_i\text{-}end(k) \wedge \neg A_j\text{-}end(k))$

As introduced in Chapter 5, Temporal Constraint 2 is in the form of simple temporal constraints, $t_i - t_j \in [lb, ub]$, in the MLQP encoding of K_{AA}. In the QSP, the temporal bounds $[lb, ub]$ are associated with event pairs. The tricky part is to identify time points in a Hybrid Flow Graph that correspond to the event pairs.

In order to identify the time points that correspond to the event pairs, I first describe an observation of a QSP. We observe that there are six cases in total for an event in a QSP. They are as follows.

1. An event is only the start event of a goal episode. See Fig. 7-4 (1).
2. An event is only the end event of a goal episode. See Fig. 7-4 (2).
3. An event is both the start event of a goal episode and the end event of another goal episode. See Fig. 7-4 (3).
4. An event is not directly connected to any goal episode, but is the first event of the QSP. See Fig. 7-5 (4).
5. An event is not directly connected to any goal episode, but is the last event of the QSP. See Fig. 7-5 (5).
6. An event is in the middle of two events. See Fig. 7-5 (6).

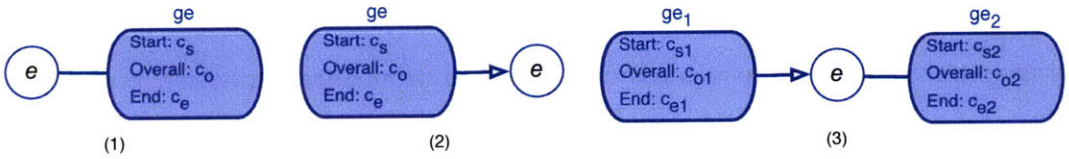


Figure 7-4: Case 1, 2, and 3 for an event in a QSP. (1) An event is the start event of a goal episode. (2) An event is the end event of a goal episode. (3) An event is both the start event of a goal episode and the end event of another goal episode.

In Case 1 - 5, the event, e, can be identified as follows.

1. Event e is the start event of goal episode ge: e is when ge starts. Suppose ge is encoded as hybrid durative action A. Then $e = t(A\text{-}start)$, where $t(A\text{-}start)$ represents the time of the fact level immediately before $A\text{-}start$.

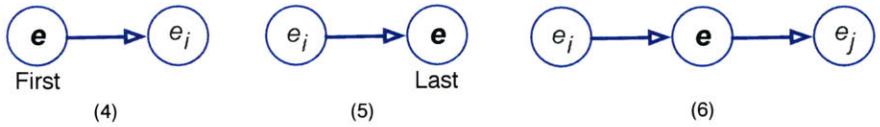


Figure 7-5: Case 4, 5, and 6 for an event **e** in a QSP. (4) An event is not directly connected to any goal episode, but is the first event of the QSP. (5) An event is not directly connected to any goal episode, but is the last event of the QSP. (6) An event is in the middle of two events.

2. Event e is the end event of goal episode g: e is when ge ends. Suppose ge is encoded as hybrid durative action A. Then $e = t(A\text{-}end)$, where $t(A\text{-}end)$ represents the time of the fact level immediately after $A\text{-}end$.

3. Event e is both the start event of goal episode ge_1 and the end event of goal episode ge_2: e is when ge_1 starts and when ge_2 ends. Suppose ge_1 is encoded as hybrid durative action A_1 and ge_2 is encoded as hybrid durative action A_2. Then $e = t(A_1\text{-}start) = t(A_2\text{-}end)$, where $t(A_1\text{-}start)$ represents the time of the fact level immediately before $A_1\text{-}start$, and $t(A_2\text{-}end)$ represents the time of the fact level immediately after $A_2\text{-}end$.

4. Event e is not directly connected to any goal episode, but is the first event of the QSP: $e = t(1)$, where $t(1)$ represents the time of the first fact level.

5. Event e is not directly connected to any goal episode, but is the last event of the QSP: $e = t(K)$, where $t(K)$ represents the time of the last fact level.

In Case 6, we observe that the three events in Fig. 7-5 (6) can be collapsed to two events without changing any of the temporal constraints on the goal episodes. I show that this can be done even when the QSP is complicated, in Fig. 7-6. This way we know that Case 6 can be transformed to one of Case 1 - 5.

Now we know how to identify the time point in a Hybrid Flow Graph that corresponds to any event e in the QSP. The following constraint in the MLQP encoding of K_{AA} represents Temporal Constraint 2:

$$\forall \text{ simple temporal constraint}\{\langle e_i, e_j\rangle, [lb, ub]\} \in \mathcal{QSP},\ e_j - e_i \in [lb, ub]. \qquad (7.1)$$

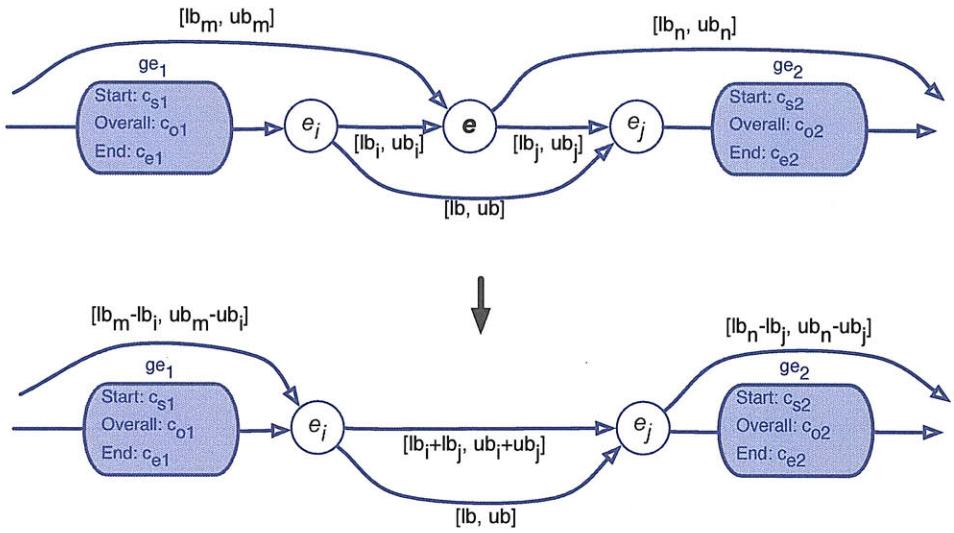


Figure 7-6: I show that the three events in Fig. 7-5 (6) can be collapsed to two events without any change in the temporal constraints on the goal episodes. The temporal bounds between events are modified accordingly.

7.4 Reformulation Correctness

In this section, I show that the reformulation is correct. In other words, the output of the reformulation is equivalent to the input of the reformulation in terms of the constraints to be enforced.

A QSP consists of two types of constraints: state constraints and temporal constraints. Each goal episode in a QSP has a set of state constraints. If the state constraints are satisfied at the right time, the goal episode is considered achieved. Temporal constraints define the order of goal episodes, and the lower and upper bounds on the time between two events in a QSP.

Suppose there are m goal episodes in a QSP. For goal episode $ge_1, ge_2, \ldots, ge_m$, $\mathrm{K_{QSP}}$ creates durative action $a_1, a_2, \ldots, a_m$, and a final goal state $\wedge_i ge_i\text{-}ended$, $i = 1, 2, \ldots, m$. In order to achieve the final goal state, each individual $ge_i\text{-}ended$ needs to be true. Because only action a_i can create the positive literal $ge_i\text{-}ended$, action a_i needs to take place, which enforces the conditions of a_i to be true. Therefore, the state constraints of each goal episode in the QSP are enforced.

Suppose goal episode ge_i has a set of predecessors ge_j, $j \in K$. $\mathrm{K_{QSP}}$ creates

durative action a_i for ge_i, and the *start* conditions of a_i include $\wedge_j ge_j$-*ended*, $j \in K$. This enforces all the predecessors to be achieved before starting goal episode ge_i. Moreover, as explained in the previous section, the first K_{QSP}-specific feature in K_{AA} ensures that, goal episodes that share the same start event or end event should start or end at the same time. Therefore, the temporal constraints that define the order of goal episodes in the QSP are enforced.

The last type of temporal constraints in a QSP we need to consider are the lower and upper bounds on the time between two events. As described in the previous section, the second K_{QSP}-specific feature in K_{AA} is to identify the time point in a Hybrid Flow Graph that corresponds to any event e in the QSP, and add to the MLQP encoding $e_j - e_i \in [lb, ub]$ for every such temporal constraint. Therefore, the temporal constraints that define the lower and upper bounds on the time between two events in the QSP are enforced.

7.5 Output Conversion

Recall from Chapter 3 that the optimal hybrid plan that is output from K_{QSP} is slightly different from the output plan from K_{DA}. I present the conversion from K_{DA}'s output to K_{QSP}'s output in this section.

The main difference between the two output plans is that, the optimal action sequence in K_{DA}'s output includes actions that are created from the goal episodes in the QSP, whereas the optimal action sequence in K_{QSP}'s output does not include such actions.

More specifically, as defined in Chapter 3, the optimal action sequence output from K_{DA}, which I call $\mathbf{A}^*_{DA}$, and the optimal action sequence output from K_{QSP}, which I call $\mathbf{A}^*_{QSP}$, are both in the form of $\{\mathbf{a}^*_{\mathbf{d}}(t_{n_1}), \mathbf{a}^*_{\mathbf{d}}(t_{n_2}), \ldots, \mathbf{a}^*_{\mathbf{d}}(t_{n_m})\}$, where $\{t_{n_1}, \ldots, t_{n_m}\} \subseteq \{t_1, \ldots, t_{N-1}\}$, $\mathbf{a}^*_{\mathbf{d}}(t_{n_i})$ is the set of $\langle a, d\rangle$ pairs at time t_{n_i}; in each pair, a is an action instantiated from the hybrid durative action types $\mathcal{DA}$ that starts at time t_{n_i}, and d is its duration. However, $\mathbf{A}^*_{DA}$ includes actions that are created from the goal episodes in the input QSP, whereas $\mathbf{A}^*_{QSP}$ omits actions that are created

from the goal episodes in the input QSP.

For example, the optimal action sequence output from K_{DA} is as follows. At time t_1, {⟨`goal-sample1`, 12⟩, ⟨`glide`, 4⟩}; at time t_2, {⟨`setRudder`, 3⟩}; at time t_5, {⟨`descend`, 3⟩, ⟨`setGulper`, 3⟩}; at time t_8, {⟨`takeSample`, 4⟩}; at time t_{13}, {⟨`goal-sample2`, 13⟩, ⟨`glide`, 5⟩}; at time t_{18}, {⟨`descend`, 3⟩}; at time t_{21}, {⟨`takeSample`, 4⟩}. In this action sequence, `goal-sample1` and `goal-sample2` are actions created from goal episodes in the input QSP.

The output conversion module in K_{QSP} is responsible for removing the actions that are created from goal episodes of the input QSP from the optimal action sequence of K_{DA}, namely from $\mathbf{A}^*_{DA}$. The pseudo code of Output Conversion in K_{QSP} is shown in Alg. 14. It enumerates the time points in $\mathbf{A}^*_{DA}$ (Line 1), and finds the actions that are created from goal episodes and removes them (Line 3-5).

Alg. 14 OutputConversion$_{QSP}$ (optimal action sequence of K_{DA}: $\mathbf{A}^*_{DA}$) **returns** optimal action sequence of K_{QSP}: $\mathbf{A}^*_{QSP}$

```
1: for t_i = t_{n_1}, ..., t_{n_m} do
2:   for each action a in a*_d(t_i) do
3:     if a is created from a goal episode then
4:       remove a from a*_d(t_i)
5:     end if
6:   end for
7: end for
8: return A*_QSP ← A*_DA
```

This chapter presented K_{QSP}, the planner that plans with qualitative state plans and hybrid durative actions with flexible durations. This completes the four approach chapters on Kongming. In the next chapter I will present empirical evaluation.

Chapter 8

Empirical Evaluation

Contents

Kongming is implemented in Java, and uses CPLEX 10.1 to solve its MLQP encoding of the hybrid planning problem. In this chapter, I first report our field demonstration of Kongming on the Odyssey IV autonomous underwater vehicle, provided by the MIT AUV Lab [auv], in the Atlantic ocean. I then describe a series of experiments in simulation, based on real-world scenarios.

8.1 Demonstration on An AUV

Our field demonstration of Kongming was in the Atlantic ocean in Cape Cod on Odyssey IV (see Fig. 8-1), an AUV designed and built by the MIT AUV Lab at Sea Grant. One of the main goals of the demonstration was to show Kongming's capability of producing optimal hybrid plans off line to achieve temporally extended goals. A

more important goal of the demonstration was to show Kongming's capability of generating mission scripts that run seamlessly on an AUV in a natural environment. Prior to this field demonstration, Odyssey IV and other AUVs at the MIT AUV Lab have only been controlled by mission scripts hand written by the engineers. Kongming automatically generates optimal plans, and enables online re-planning. The online re-planning demonstration is not recorded in this thesis, but can be found at [rep].

Figure 8-1: Odyssey IV, an AUV from the MIT AUV Lab at Sea Grant, is $0.7 \times 1.4 \times 2.2$m, weighs 500kg dry and about 1000kg wet, and has rated depth 6000m. More information about the vehicle can be found in [ody].

As this was our first field demonstration, although Kongming is fully capable of planning in 3D, for safety reasons, the vehicle was kept on the surface during the tests, in order to maintain constant communication. As the vehicle was limited to operate on the surface, this reduced planning to one continuous action type, namely the `GoToWaypoint` behavior. Moreover, at the time of the demonstration, the vehicle could not perform any discrete action, so `GoToWaypoint` was the only hybrid action type available for the field demonstration.

I conducted two major tests. The Test 1 scenario consists of reaching three waypoints before going to the pickup point. The graphical user interface (GUI) used by Kongming for the scenario is shown in Fig. 8-3. As we can see, an obstacle prevents Odyssey IV from following a straight line from waypoint 2 to 3. The obstacle is

imaginary in the ocean. I use a Visibility Graph (VG) path planner to pre-process obstacle avoidance. More specifically, when two waypoints are separated by an obstacle, the VG path planner identifies and inserts some of the obstacle corners to form the shortest path between the two waypoints to go around the obstacle.

Note that Kongming is capable of avoiding obstacles, by encoding this requirement as external constraints $\mathcal{C}$. The constraint encoding was introduced in Section 5.3.2. In this deployment, I use a VG path planner to pre-process obstacle avoidance. The purpose is to avoid the classic pitfall in discrete-time constraint-based path planning, where "outside of obstacles" is only guaranteed at each time step but not in between the time steps. Hence, the resulting path may "cut the corners" of the obstacle. I show an example of a corner-cutting path, compared with a VG-pre-processed path in Fig. 8-2. The blue path shows the path when using the VG pre-processing, and the red path shows the path without the VG pre-processing. As we can see, along the red path, the position at time step i and the position at time step $i+1$ are outside the obstacle, but the path in between the two time steps is undesirably inside the obstacle.

The QSP input to Kongming is shown in Fig. 8-4. As we can see, two goal episodes, ge_3 and ge_4, to reach the obstacle corners are added. As the vehicle is limited to operate on the surface, this reduces planning to one continuous action type, namely the `GoToWaypoint` behavior. At the time of the demonstration, the vehicle could perform no discrete action, so `GoToWaypoint` was the only hybrid action type in this field demonstration. The initial condition for Kongming is to be at the start point, as seen in Fig. 8-3. The external constraints $\mathcal{C}$ are empty, because obstacles are avoided using VG path planner, and there are no other external constraints. The objective function is a quadratic function to minimize the total distance traveled.

The mission script generated by Kongming is comprised of 22 behaviors, and is listed in Appendix A. The mission script commands Odyssey IV to go to the waypoints specified in the QSP as well as all the intermediate waypoints generated by Kongming. The behavior-based controller of the vehicle then maneuvers the vehicle to those waypoints within specified time bounds. The trajectory of the vehicle is shown in

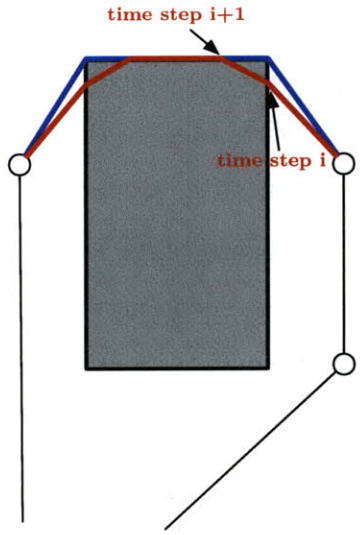


Figure 8-2: Comparison of an example of a corner-cutting path with a VG-pre-processed path. The blue path shows the path when using the VG pre-processing, and the red path shows the path without the VG pre-processing. As we can see, along the red path, the position at time step i and the position at time step $i+1$ are outside the obstacle, but the path in between the two time steps is undesirably inside the obstacle.

Fig. 8-5, where the wavy parts were due to currents in the ocean and the fact that the behavior-based controller was under-tuned. The extra leg close to the bottom of the plot occurs because at the start of the mission, the vehicle was not at the assumed start location. Hence it had to traverse to the assumed start location first.

Test 2 covers a larger area in the ocean and is a longer mission than Test 1. Test 2 shows that Kongming performs efficiently when I scale up the problem size in this scenario. It took Kongming 0.455 seconds to generate an optimal plan for Test 2, while it took Kongming 0.319 seconds to generate an optimal plan for Test 1 on the same machine and with the same CPLEX solver.

The scenario consists of reaching 10 waypoints before the pickup point, in order to draw the letter K (for Kongming), as shown in the GUI in Fig. 8-6. No obstacle is involved. The QSP input to Kongming, as shown in Fig. 8-7, is different from that of Test 1. The rest of the input to Kongming is the same as in Test 1.

The mission script generated by Kongming is comprised of 37 behaviors, and is listed in Appendix A. The mission script commands Odyssey IV to go to the waypoints

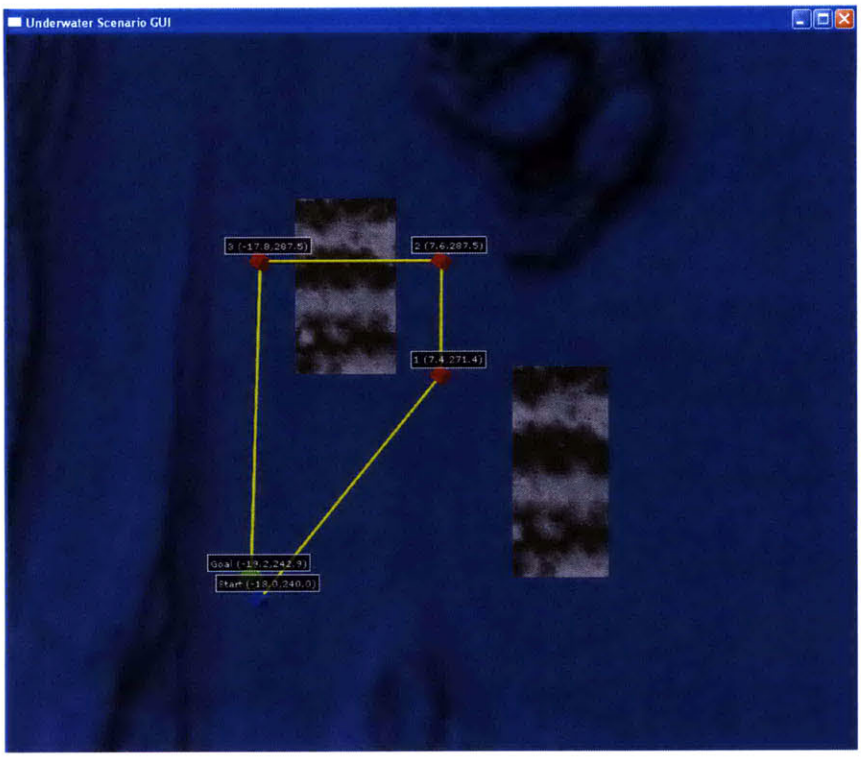


Figure 8-3: Display of Kongming's graphical user interface for the scenario of Test 1. The mission requires Odyssey IV to reach three waypoints before going to the pickup point. There is an obstacle between Waypoint 2 and 3.

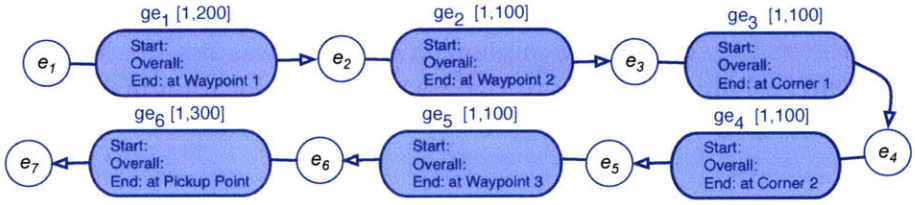


Figure 8-4: The QSP for Test 1. Each goal episode in the QSP specifies the state of reaching a waypoint. The start point:(-18,240), Waypoint 1:(7.3,271.3), Waypoint 2:(7.5,287.3), Obstacle Corner 1:(1.3,296.2), Obstacle Corner 2:(-12.9,296.2), Waypoint 3:(-17.7,287.5), the Pickup Point:(-19.1,242.9). The temporal constraint for each goal episode is specified in square brackets.

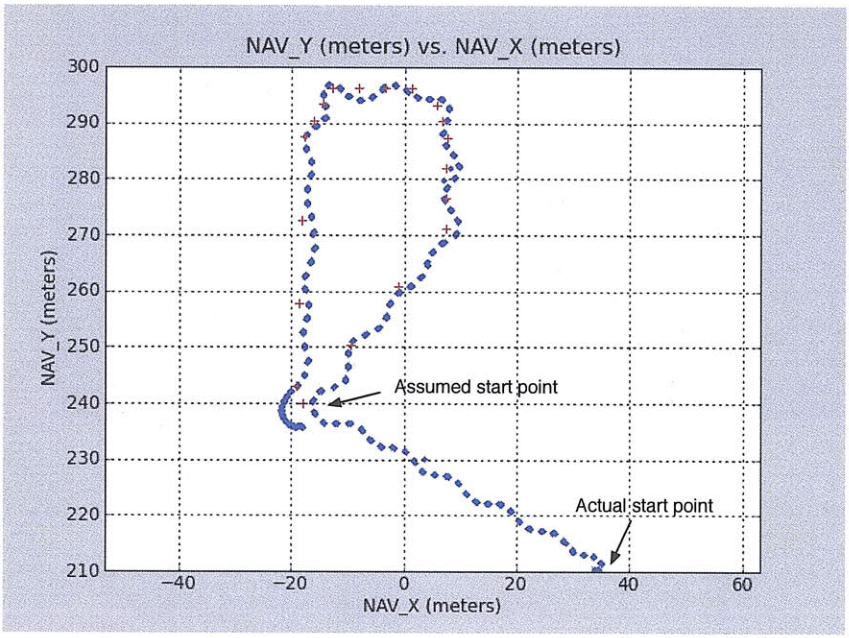


Figure 8-5: The plot of the actual trajectory of Odyssey IV executing the mission script generated by Kongming. AUV's trajectory is in blue dots, and the waypoints planned by Kongming are in red crosses. The extra leg close to the bottom of the plot occurs because at the start of the mission, the vehicle was not at the assumed start location. Hence it had to traverse to the assumed start location first.

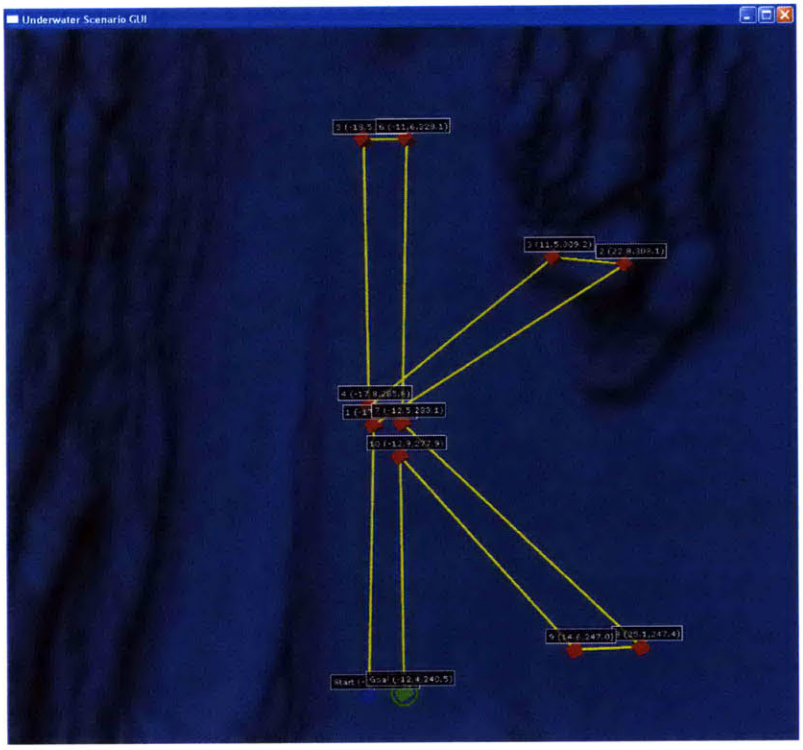


Figure 8-6: Display of Kongming's graphical user interface for the scenario of Test 2. The mission requires Odyssey IV to reach 10 waypoints before going to the pickup point, in order to spell the letter K (the first letter in Kongming).

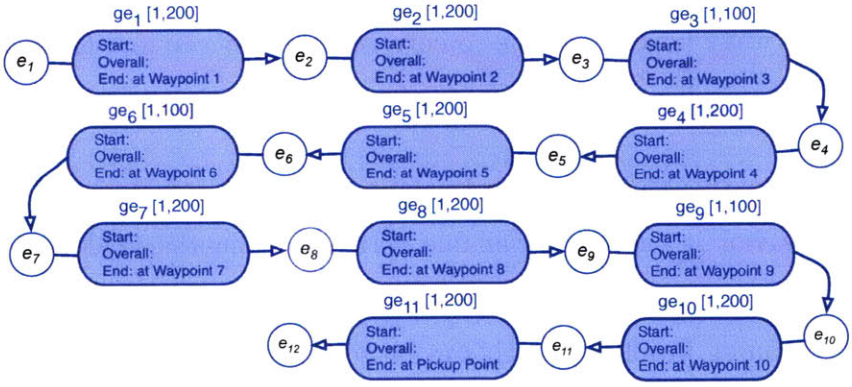


Figure 8-7: The QSP for Test 2. Each goal episode in the QSP specifies the state of reaching a waypoint. The start point:(-18,240), Waypoint 1:(-17.7,283.1), Waypoint 2:(22.8,308.1), Waypoint 3:(11.5,309.2), Waypoint 4:(-17.8,285.8), Waypoint 5:(-18.5,328.1), Waypoint 6:(-11.6,328.1), Waypoint 7:(-12.5,283.1), Waypoint 8:(25.1,247.4), Waypoint 9:(14.6,247.0), Waypoint 10:(-12.9,277.9), and the Pickup Point:(-12.4,240.5). The temporal constraint for each goal episode is specified in square brackets.

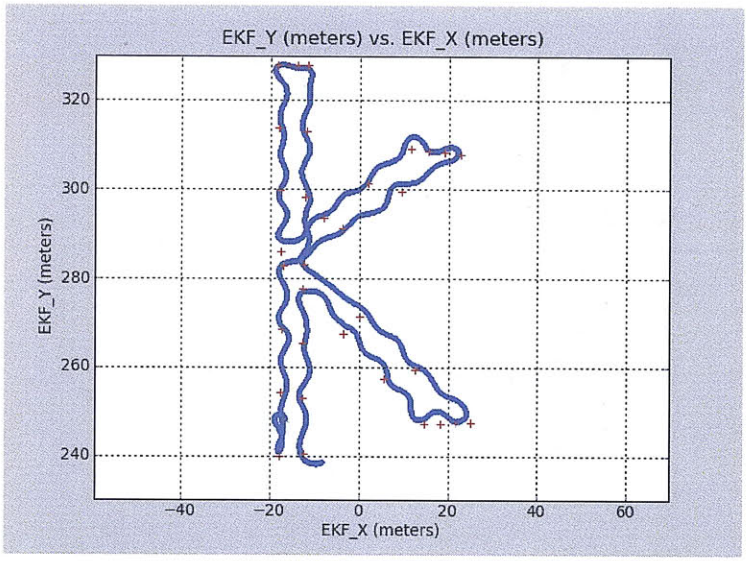


Figure 8-8: The plot of the actual trajectory of Odyssey IV executing the mission script generated by Kongming. AUV's trajectory is in blue dots, and the waypoints planned by Kongming are in red crosses.

specified in the QSP as well as all the intermediate waypoints generated by Kongming. The behavior-based controller of the vehicle then maneuvers the vehicle to those waypoints within specified time bounds. The trajectory of the vehicle is shown in Fig. 8-8, where the wavy parts were due to currents in the ocean and the fact that the behavior-based controller was under-tuned.

Note that due to the limitations and safety constraints of Odyssey IV, the `GoToWaypoint` behavior is the only available action type in the field demonstration. This reduces the hybrid QSP generative planning problem that Kongming solves to a QSP path planning problem, which can be solved by path planners, such as Sulu [LW05a, Léa05]. Although this field demonstration did not showcase Kongming's full capability, it demonstrated Kongming's capability of generating mission scripts that run seamlessly on an AUV in a natural environment. As mentioned before, prior to this field demonstration, all AUVs at the MIT AUV Lab have only been controlled by mission scripts hand written by the engineers. Kongming automatically generates optimal plans, and enables online re-planning.

8.2 Experimental Results

I tested Kongming in simulation in two domains: the underwater vehicle domain and the air vehicle domain. In the underwater domain, I used scenarios adapted from MBARI science missions. In the air vehicle domain, I used fire fighting scenarios, which are variants of the scenario described in Section 3.1.

I tested Kongming in the following three aspects. First, in order to show the benefit of Kongming's compact plan representation, the Hybrid Flow Graph, I developed a direct constraint encoding of the hybrid planning problem, which bypasses the Hybrid Flow Graph. I compared the performance of Kongming with that of the direct encoding. Second, I showed how Kongming scales computationally in terms of the length of the time increment Δt, which is the length of each action level in the Hybrid Flow Graph. Finally, I showed how Kongming scales computationally in terms of the number of hybrid action types. More specifically, I compared Kongming's scalability in the number of hybrid action types with specified dynamics with Kongming's scalability in the number of hybrid action types without specified dynamics.

8.2.1 Benefit of Hybrid Flow Graph Compared with Direct MLQP Encoding

As mentioned in Section 1.3, a straightforward way to solving the hybrid planning problem is to encode the problem directly into a mixed logic quadratic programming (MLQP) problem, and solve it using a state-of-the-art MLQP solver. Doing this, however, puts all the computational burden on the solver. Our experiment in this section shows that the search space is prohibitive for even small problems, by comparing the performance of Kongming with that of the direct encoding planner, which I call $\text{Planner}_{\text{DE}}$.

The main difference between Kongming and $\text{Planner}_{\text{DE}}$ is the Hybrid Flow Graph, which Kongming uses to represent the hybrid plan space. As introduced in Chapter 4, the Hybrid Flow Graph builds upon the Planning Graph [BF97], and captures all feasible continuous state trajectories using flow tubes. The flow tube computation in

essence provides projection of reachable states into the future. The main benefit of this projection is to help identify mutually exclusive actions whose continuous preconditions or continuous state trajectories are mutually exclusive. As shown in [BF97], identifying mutual exclusion relations is largely useful in reducing the search space.

$\text{Planner}_{\text{DE}}$ encodes directly the hybrid planning problem that K_{AA} solves. Its encoding rules are listed as follows.

1. Initial conditions are satisfied;
2. Goal conditions are satisfied;
3. If an action takes place at time i, then its preconditions are true at i, and its discrete effects are true at $i + 1$;
4. If an action with specified dynamics takes place at time i, then its control variables at i are within the control limits, and its state variables and control variable satisfy the state transition equation;
5. If a propositional fact is true at time $i + 1$, then at least one of the actions that have the fact as an effect takes place at i;
6. If the continuous state changes from time i to time $i + 1$, then at least one of the actions with specified dynamics takes place at i;
7. If two propositional facts negate each other, then they are *statically mutually exclusive* and cannot be true at the same time;
8. If one action's effect or precondition negates the other action's effect or precondition, or if the continuous preconditions of two actions have an empty intersection, then the actions are *statically mutually exclusive* and cannot take place at the same time;
9. External constraints are satisfied;
10. Durative actions' duration bounds are satisfied;

11. QSP temporal constraints are satisfied.

The main difference of the direct encoding from Kongming's approach introduced in Chapter 4 and 5 is the mutual exclusion relations. There are two aspects to the mutual exclusion relations. First, in the direct encoding, there are only static mutual exclusions, for example, two propositions that negate each other are always mutually exclusive. They are called *static* because their mutual exclusion relations never change. In contrast, there are not only static but also non-static mutual exclusions in Kongming's approach. The non-static mutual exclusions may change over time. For example, two facts are mutually exclusive if all actions that create one fact are mutually exclusive with all actions that create the other fact. Because more actions may become available for creating the facts as time elapses, at a different time step, the mutual exclusion relation may disappear. The second aspect is that, in the direct encoding, mutual exclusions among facts are limited to propositions, whereas in Kongming's approach, mutual exclusions among facts also include continuous regions. This is enabled by flow tube computation in the Hybrid Flow Graph expansion. In summary, the differences in mutual exclusion relations contribute to the main difference between the direct encoding approach and Kongming's approach. I show their difference in performance as follows.

I compare the performance on two sets of scenarios: a set of underwater scenarios and a set of fire fighting scenarios. The corresponding input files are given in Appendix B.

- **Underwater Scenario 1** involves that an underwater vehicle goes to a specific region on the surface of the ocean to take a sample. The world is 2D and within a square map. There is one goal in the QSP, one action type with specified dynamics, `glide`, and one action type without specified dynamics, `take sample`. The objective is to minimize distance traveled.

- **Underwater Scenario 2** involves that an underwater vehicle goes to two specific regions on the surface of the ocean to take samples. The world is 2D and within a square map. There are two goals in sequence in the QSP, one action

type with specified dynamics, `glide`, and one action type without specified dynamics, `take sample`. Note that the location for `take sample` can be either of the two regions. The objective is to minimize distance traveled.

- **Underwater Scenario 3** involves that an underwater vehicle goes to a specific region in the ocean to take a sample. The world is 3D and there is a dangerous water column in the middle that the vehicle needs to avoid. There is one goal in the QSP, three action types with specified dynamics, `glide`, `descend`, `ascend`, and one action type without specified dynamics, `take sample`. The objective is to minimize distance traveled.

These three scenarios are adapted from the MBARI survey missions, and are representative of part of the real missions.

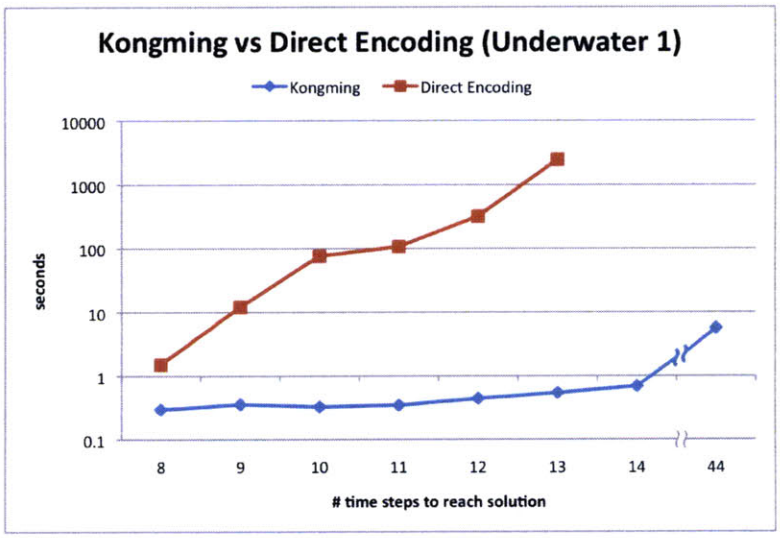


Figure 8-9: Logarithmic scale plot of the result of Underwater Scenario 1. The horizontal axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. Blue line shows the performance of Kongming. Red line shows the performance of $\text{Planner}_{\text{DE}}$. Kongming scales well. At time step 44, the computation time is 5.677 sec. However, after time step 13, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours, which is 1.728E+05 seconds.

The logarithmic scale plot of the result of Underwater Scenario 1 is shown in Fig. 8-9. The x axis is the number of time steps to reach solution. The vertical axis is computation time in seconds.

The number of time steps in the x axis means the following. If there are n time steps, in $\text{Planner}_{\text{DE}}$ it means that the last time index is $n+1$, as the initial time index is 1; and in Kongming it means that the last fact level is Level $n+1$, as the initial fact level is Level 1. When I shrink the velocity range of the action types and keep the rest of the problem unchanged, more time steps need to be taken in order to reach the same goals. Increasing the number of time steps increases the complexity of the hybrid planning problem, because there are more variables and constraints in the MLQP encoding, and for Kongming the Hybrid Flow Graph is expanded into a larger graph.

As we can see from the plot, Kongming scales well. At time step 44, the computation time is 5.677 sec. However, after time step 13, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

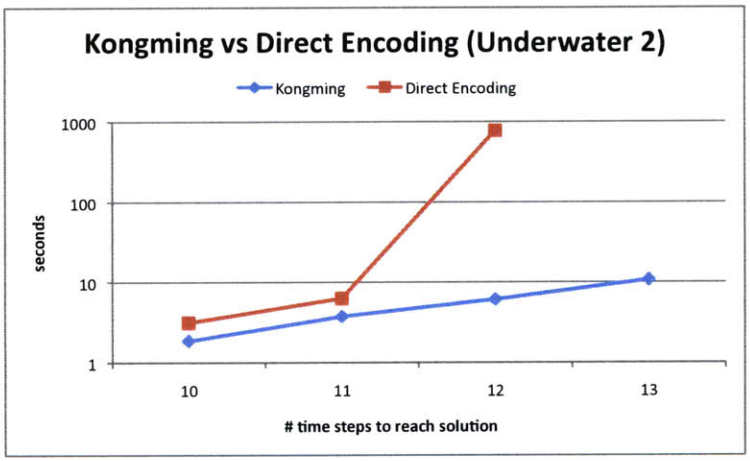


Figure 8-10: Logarithmic scale plot of the result of Underwater Scenario 2. The horizontal axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. Blue line shows the performance of Kongming. Red line shows the performance of $\text{Planner}_{\text{DE}}$. Kongming scales well. However, after time step 12, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

The logarithmic scale plot of the result of Underwater Scenario 2 is shown in Fig. 8-10. The x axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. As we can see from the plot, Kongming scales well. However, after time step 12, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

The logarithmic scale plot of the result of Underwater Scenario 3 is shown in

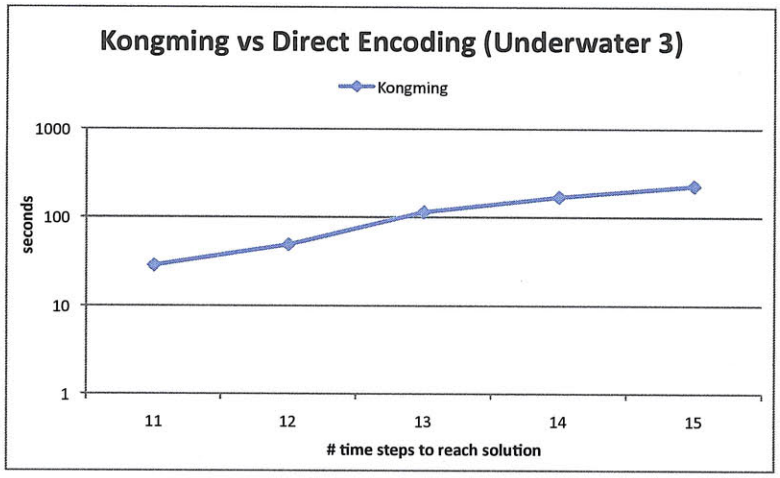


Figure 8-11: Logarithmic scale plot of the result of Underwater Scenario 3. The horizontal axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. Blue line shows the performance of Kongming. Kongming scales well. However, $\text{Planner}_{\text{DE}}$ cannot solve any instances within 48 hours.

Fig. 8-11. The x axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. As we can see from the plot, Kongming scales well. However, $\text{Planner}_{\text{DE}}$ cannot solve any instances within 48 hours.

The following two scenarios are adapted from fire fighting missions in [Léa05], and closely resemble the original missions.

- **Fire-fighting Scenario 1** involves that an air vehicle goes to a fire region and extinguishes the fire. To put out the fire, the vehicle needs to have water, which it can take from a lake. After the fire is extinguished, the vehicle needs to take a photo. The world is 2D as we assume the air vehicle flies at a constant altitude. There is one goal in the QSP, one action type with specified dynamics, `fly`, and two action types without specified dynamics, `fill water` and `take photo`. The objective is to minimize distance traveled.

- **Fire-fighting Scenario 2** involves that an air vehicle goes to extinguish two fires. To put out the fire, the vehicle needs to have water, which it can take from a lake. After the fires are extinguished, the vehicle needs to take photos. The world is 2D with a no-fly zone to avoid. There are two goals in the QSP,

one action type with specified dynamics, `fly`, and two action types without specified dynamics, `fill water` and `take photo`. Note that the location for `take photo` can be either of the two fire regions. The objective is to minimize distance traveled.

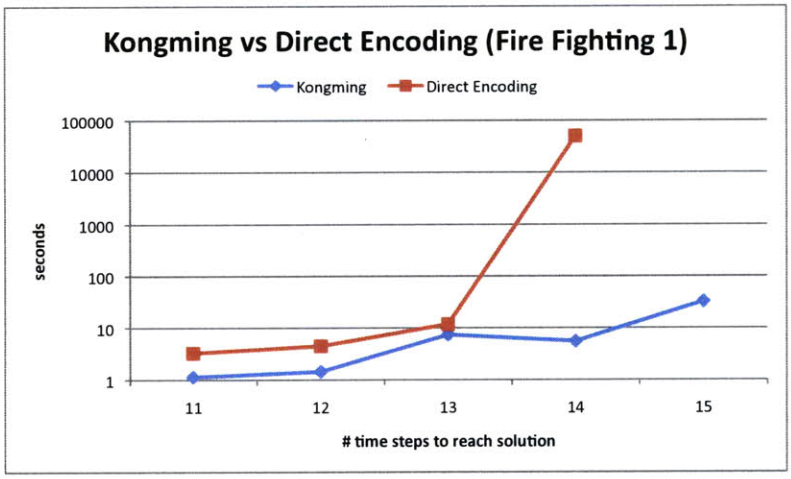


Figure 8-12: Logarithmic scale plot of the result of Fire-fighting Scenario 1. The horizontal axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. Blue line shows the performance of Kongming. Red line shows the performance of $\text{Planner}_{\text{DE}}$. Kongming scales well. However, after time step 14, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

The logarithmic scale plot of the result of Fire-fighting Scenario 1 is shown in Fig. 8-12. As we can see from the plot, Kongming scales well. However, after time step 14, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

The logarithmic scale plot of the result of Fire-fighting Scenario 2 is shown in Fig. 8-13. As we can see from the plot, Kongming scales well. However, after time step 23, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

To summarize, the non-static and the continuous fact mutual exclusion relations are essential in reducing the search space, and in order to identify a significant portion of such mutual exclusion relations, flow tubes need to be computed to propagate the reachable continuous state regions and expand the Hybrid Flow Graph.

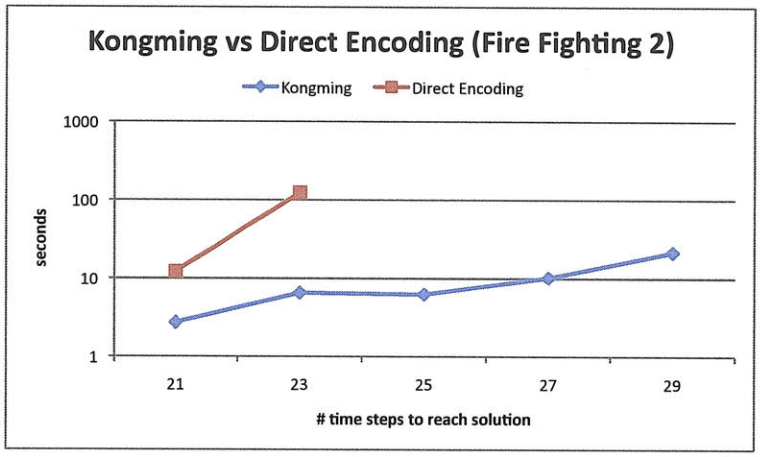


Figure 8-13: Logarithmic scale plot of the result of Fire-fighting Scenario 2. The horizontal axis is the number of time steps to reach solution. The vertical axis is computation time in seconds. Blue line shows the performance of Kongming. Red line shows the performance of $\text{Planner}_{\text{DE}}$. Kongming scales well. However, after time step 23, the computation time of $\text{Planner}_{\text{DE}}$ goes beyond 48 hours.

8.2.2 Scaling in Δt

Recall from Chapter 6 that Kongming discretizes the flow tube of a durative action into "slices", each of which is Δt long in time. This Δt is also the duration of each action level in the Hybrid Flow Graph. As discussed in Section 6.2.2, it is desirable to set Δt equal to the time increment in the state transition equation of a hybrid action type, $\forall t_i,\ \mathbf{x}(t_i) = \mathbf{A}\mathbf{x}(t_{i-1}) + \mathbf{B}\mathbf{u}(t_{i-1})$, because this ensures that the flow tube computation for each hybrid atomic action is as accurate as the state equation. However, for agile dynamic systems, such as an air vehicle, whose time increment in the state transition equation can be as small as 0.1 seconds, Kongming will need to construct a fairly large Hybrid Flow Graph. If a planning problem has a horizon of half an hour, there are 18000 levels in the Hybrid Flow Graph, which is not computationally realistic. Hence, the strategy Kongming takes is to set Δt as small as the computation power allows.

In this section, I tested how Kongming scales in terms of the value of Δt. I used Fire-fighting Scenario 1 in this test, and randomly generated 10 problems from the scenario, by altering the parameters, such as initial location, lake location and fire location. In each of the 10 problems I decreased Δt from 1 second to 0.1 second, which corresponds roughly to from 20 to 200 action levels in the Hybrid Flow Graph

on average. I recorded the time to expand the Hybrid Flow Graph and the time to solve the MLQPs in logarithmic scale in Fig. 8-14. For each Δt value, I recorded the average computation time over the 10 problems. The computation time has two components. One is the time spent on expanding the Hybrid Flow Graph, and the other is the time spent by CPLEX on solving the MLQPs.

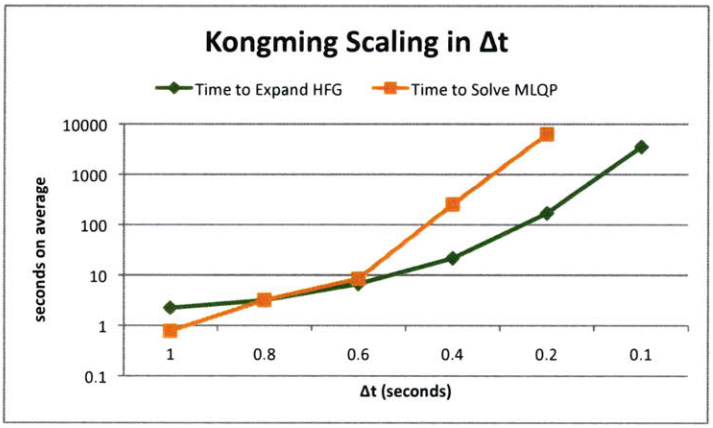


Figure 8-14: On the horizontal axis, Δt is the length of a time step and the temporal length of each action level. Δt is decreased while the rest of the problem is unchanged. The vertical axis shows the average computation time in seconds in logarithmic scale. Green line shows the time to expand the Hybrid Flow Graph. Orange line shows the time to solve the MLQPs. As Δt decreases, the time to solve MLQPs and the time to expand the Hybrid Flow Graph both increase. The time to solve MLQPs grows 1 to 2 orders of magnitude faster than the time to expand the graph.

We can see from the plot, as Δt decreases, the time to expand the graph and the time to solve the MLQPs both increase. The time spent by CPLEX on solving the MLQPs increases faster than the time to expand the graph. When Δt is smaller than 0.2, the time to solve the MLQPs exceeds 48 hours. When $\Delta t = 0.2$, the average number of action levels in the Hybrid Flow Graph is 103.

8.2.3 Scaling in The Number of Action Types

In order to test how the number of hybrid durative action types affects the performance of Kongming, I randomly generated 70 problems based on Underwater Scenario 3. Recall that I call hybrid action types with specified dynamics, *continuous*, for example, action `descend`. I call hybrid action types without specified dynamics, *discrete*, for

example, action `take sample`. In this test, I first increase the number of continuous action types, while maintaining 3 discrete action types. Second, I increase the number of discrete action types, while maintaining 3 continuous action types. Note that they are action *types*, not action instances.

Among the 70 problems, there are 5 problems for each combination of continuous action type number and discrete action type number. The 5 problems in each set contain different parameters, such as initial, goal location, the continuous preconditions and durations of actions. The average computation time in seconds in logarithmic scale is recorded over the 5 problems for each set in Fig. 8-15.

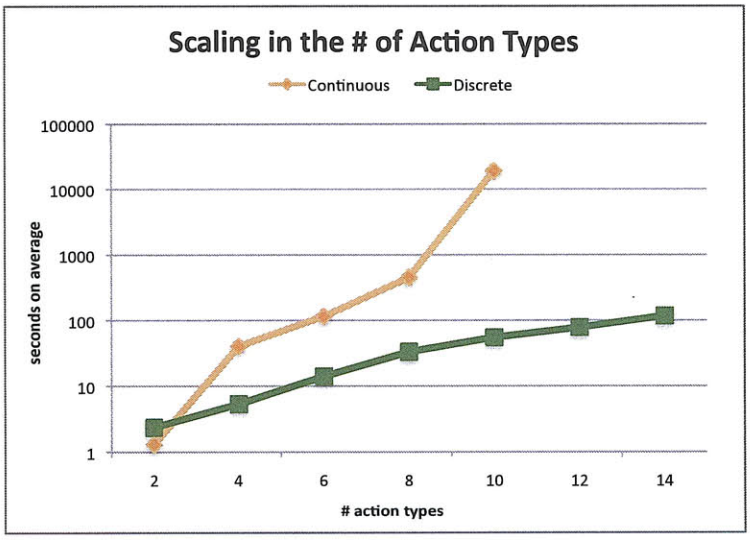


Figure 8-15: Logarithmic plot of computation time in seconds for different number of continuous action types, and different number of discrete action types. Green line shows how Kongming scales as the number of discrete action types increases, while maintaining 3 continuous action types. Orange line shows how Kongming scales as the number of continuous action types increases, while maintaining 3 discrete action types.

We can see from Fig. 8-15, Kongming scales better in terms of the number of discrete action types than in terms of the number of continuous action types. With more than 10 continuous action types, the computation time of Kongming exceeds 48 hours. Note that 99% of the computation time is spent on solving the MLQPs by CPLEX in this case. The reason for it is that in each action level, for one continuous action type, there can be a large number of flow tubes based on different initial regions,

which then propagate to a large number of continuous facts in the next fact level, and so on. This significantly increases the number of variables and constraints in the MLQPs.

However, the good news is that in real-world problems, there are only a small number of continuous action types involved, because different continuous action types correspond to different continuous dynamics of the autonomous system. For example, an AUV has three continuous action types: `glide`, `descend` and `ascend`.

Chapter 9

Conclusion

Contents

In this thesis, I have presented a hybrid generative planner, called Kongming. It is capable of planning for temporally extended goals for hybrid autonomous systems. Kongming elevates the level of commanding, such that human operators only need to specify a set of time evolved goals that they want to accomplish, and have the planner itself produce the series of actions that achieve the mission goals, based on the model of the physical system under control.

9.1 Summary

In this section, I summarize the chapters in the thesis. Chapter 1 motivates the hybrid generative planning problem, and gives an overview of the whole thesis. Chapter 2 relates Kongming with prior work, in terms of the problem representation language,

continuous planning, and planning for temporally extended goals. Chapter 2 also reviews prior work that Kongming builds upon. Chapter 3 formally defines the 3 planning problems: the problem for $\mathbf{K_{QSP}}$, the problem for $\mathbf{K_{DA}}$, and the problem for $\mathbf{K_{AA}}$. Recall that Kongming is divided into three planners. K_{QSP} plans for QSPs, by reformulating the QSP to durative actions and a final goal state, and then engaging K_{DA}. K_{DA} plans with hybrid durative actions with flexible durations, by reformulating durative actions to atomic actions and then engaging K_{AA}. Chapter 4 introduces the compact plan representation for $\mathbf{K_{AA}}$, called the Hybrid Flow Graph. It includes the flow tube representation of actions, as well as the definition and construction of the Hybrid Flow Graph. Chapter 5 introduces the constraint-based planner in K_{AA}. It includes the algorithm architecture and the mixed logic quadratic program (MLQP) constraint encoding of the Hybrid Flow Graph. Chapter 6 describes how the K_{DA} planner plans for durative actions. It reformulates durative actions to atomic actions and then engages K_{AA}. Chapter 7 describes how the K_{QSP} planner plans for QSPs. It reformulates the QSP to durative actions and a final goal state, and then engages K_{DA}. Chapter 8 presents the empirical evaluation of Kongming.

9.2 Future Work

There are several interesting directions for future work.

9.2.1 Responding to Disturbances and Uncertainty

An important goal of having a generative planner for autonomous systems is to react to the dynamically changing environment, noise in the dynamical system, or plan execution failure. For example, during the execution of an underwater mission, the path of an AUV may deviate from the planned path due to currents or system noise. This error aggregates over time, and can cause plan execution failure. If the planner outputs a spatially and temporally flexible plan instead of a rigid path, then this situation can be effectively avoided. In a different case, where goals change, new obstacles are discovered or the original plan fails to complete, the planner should be

able to re-plan online based on the new situation. We next discuss these two cases in the following subsections.

Qualitative Control Plan Output

In real-world missions, there exist disturbances in the environment and noise in the dynamical system. Hence, the actual path in execution may deviate from the planned path. The error generally aggregates over time, and can lead to execution failure. Flow tubes are a natural solution to this error, because they represent flexible valid operating regions in state space, rather than a single trajectory.

Currently Kongming uses flow tubes only for planning. Recall the description in Chapter 4, Kongming builds the hybrid flow graph one level by one level by connecting together flow tubes that represent hybrid actions. Then Kongming searches for a valid and optimal plan by encoding the hybrid flow graph as an MLQP and solving it. The solution to the MLQP corresponds to a trajectory going through the chain of flow tubes in the hybrid flow graph. The problem with this output of Kongming is that it is rigid and cannot react to disturbances.

A natural extension to Kongming is to output a chain of flow tubes, instead of a rigid trajectory. This chain of flow tubes is also called a Qualitative Control Plan (QCP), as appeared in Hofmann's work [Hof06]. Thus flow tubes will also be used in execution, and their spatial and temporal flexibility will be greatly beneficial to real-world missions in a dynamic and uncertain environment. As described in [Hof06], so long as the executed trajectory of a vehicle stays within the flow tubes, a valid control policy is guaranteed to exist, and the plan will execute successfully.

Incremental Re-planning

When goals change, new obstacles are discovered or the original plan fails to complete during execution, it is desirable that the planner is capable of re-planning online based on the new situation. Currently Kongming is capable of preliminary re-planning. In other words, whenever re-planning is needed, Kongming takes the current state of the autonomous system as the new initial condition, the new temporally extended goal,

which excludes the goals that have already been achieved from the original temporally extended goal, and the new external constraints due to environment changes. The rest of the input to Kongming remains unchanged from the original input. Then Kongming solves the updated planning problem from scratch.

Re-planning from scratch is not very efficient, as the information obtained previously from expanding the Hybrid Flow Graph and encoding and solving the MLQPs is lost every time re-planning takes place. In most real-world scenarios, there is little time allowed for re-planning. For example, when an air vehicle detects an unexpected storm in front blocking its planned route, it needs to re-plan fast enough to avoid the storm and still reach the original goals with as little detour as possible. Under tight timing constraints for re-planning, an incremental version of Kongming would be highly desirable.

9.2.2 Heuristic Forward Search

Recall that the core of Kongming (K_{AA}) consists of the following two parts: 1) a compact representation of the space of possible hybrid plans, which can be employed by a wide range of planning algorithms, from constraint-based planning to heuristic search planning; and 2) a constraint-based planning algorithm that operates on the hybrid plan representation.

Heuristic forward search planners [CCFL09, CFLS08a, HN01, MBB+09, DK01b, BG99, BG01, McD96] perform search directly on the plan representation using heuristics obtained from relaxation. The strength of heuristic search planners is that heuristics estimate the cost to goal to guide the search for a valid plan. I focus on a recent heuristic search planner, COLIN [CCFL09], as it is the most close to Kongming with respect to the type of hybrid planning problems it solves. COLIN was compared with Kongming in Chapter 2. In this section, I review briefly the heuristic search approach in COLIN.

COLIN extends the temporal relaxed planning graph (TRPG) heuristic of CRIKEY3 [CFLS08b] to handle problems with continuous numeric effects. CRIKEY3 uses a Simple Temporal Network (STN) to capture the temporal constraints. By constructing

a STN at each state in the search space, CRIKEY3 ensures the action choices are temporally consistent. CRIKEY3 employs a TRPG heuristic to navigate the search space. In the TRPG, delete effects are ignored, as in classical RPG (relaxed planning graph). Moreover, the durations of actions are used to offset start and end points between fact levels in order to capture the temporal constraints on the start and the end of actions. To provide guidance in problems with continuous numeric effects, COLIN adds bounds on metric variables in the TRPG, encodes the linear continuous effects as the start effects of actions, and attaches constraints on continuous resource consumption to the end points of the relevant actions.

It would be interesting future work to study the possibilities of applying the extended TRPG in COLIN to the plan representation in Kongming.

9.2.3 Explanation for Plan Failure

When no solution exists for a planning problem, Kongming fails to find a valid plan. In this situation, more often than not, we would like to know the reason. Such explanation for plan failure is useful in guiding the user to change the planning problem specification, so that the problem becomes solvable. For example, if the user knows that the problem is unsolvable because the autonomous system cannot reach some of the goals within the specified temporal constraints, then the user can loosen the temporal constraints or remove some of the goals.

When Kongming solves a hybrid planning problem, there are two conditions under which no valid plan exists. One is when the goals cannot be all contained in the Hybrid Flow Graph. The other is when the mixed logic quadratic program (MLQP) constraint encoding of the Hybrid Flow Graph cannot be solved. The second condition is of particular interest. In this case, the problem of finding explanation for plan failure is the problem of finding the minimal set of constraints in the MLQP that contribute to the infeasibility. There has been a lot of work in finding the minimal infeasible set or the minimal conflict set from a linear program (LP), a mixed integer linear program (MILP) or a disjunctive linear program (DLP) [Chi97, PR96, LW05b]. It would be interesting to apply these methods to Kongming's constraint encoding to

identify reason for plan failure.

9.3 Contributions

This thesis has the following contributions:

- I identified and formally defined the hybrid generative planning problem, which widely exists in the real world in controlling autonomous systems.
- I developed a novel approach to solve the hybrid generative planning problem for temporally extended goals. There are two main innovations in the approach:
    1. A compact representation of the hybrid plan space, called a Hybrid Flow Graph. It provides a natural way of representing continuous trajectories in a discrete planning framework. It combines the strengths of a Planning Graph for discrete actions and Flow Tubes for continuous actions.
    2. Novel reformulation schemes to handle temporally flexible actions and qualitative state plans. They reformulate durative temporally flexible actions to atomic actions, and reformulate qualitative state plans to durative actions and a final goal state.
- I implemented a general-purpose planner, called Kongming. It enables first, simple and natural commanding of autonomous systems by humans, and second, re-planning to react to disturbances and failure.
- I successfully demonstrated Kongming's capability of generating mission scripts that run seamlessly on an autonomous underwater vehicle in the ocean. I also conducted a range of experiments based on real-world scenarios in simulation, and showed that the plan representation in Kongming, the Hybrid Flow Graph, helps reduce the computation time on average by over 3 orders of magnitude.

List of Figures

Appendix A

A.1 Mission Script for Test 1

The mission script generated by Kongming for Test 1 in Section 8.1 is the following.

Name = test1

```
Task = FunctionGenerator
{
    Priority    = 3
    Name        = Z-axis
    TimeOut     = 600
    InitialState    = ON
    ErrorFlag   = EndMission
    FUNCTION    = Constant
    DOF     = HEAVE
    Magnitude   = 15
}
Task = GoToWaypoint
{
    Priority    = 3
    Name        = GoToWaypoint
    TimeOut     = 600
    InitialState    = ON
```

```
    CompleteFlag = Atwaypoint0
    TimeoutFlag = Stop
    ErrorFlag   = Stop
    Location    = -18, 240
    Radius    = 3.0
    Speed     = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track1
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint0
  CompleteFlag = Atwaypoint1
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -9.5608294, 250.4187246666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track2
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint1
  CompleteFlag = Atwaypoint2
  TimeoutFlag = Stop
```

```
  ErrorFlag = Stop
  Location = -1.1216587999999978, 260.8374493333333
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track3
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint2
  CompleteFlag = Atwaypoint3
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 7.317511800000001, 271.256174
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track4
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint3
  CompleteFlag = Atwaypoint4
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 7.365495466666668, 276.62060833333334
```

```
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track5
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint4
  CompleteFlag = Atwaypoint5
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 7.413479133333334, 281.9850426666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track6
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint5
  CompleteFlag = Atwaypoint6
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 7.461462800000001, 287.34947700000004
  Radius = 3.0
  Speed = 15.0
```

```
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track7
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint6
  CompleteFlag = Atwaypoint7
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 6.533150400000001, 290.30701600000003
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track8
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint7
  CompleteFlag = Atwaypoint8
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 5.604838000000001, 293.264555
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
```

```
{
  Priority = 3
  Name = Track9
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint8
  CompleteFlag = Atwaypoint9
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 1.2608950000000005, 296.222094
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track10
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint9
  CompleteFlag = Atwaypoint10
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -3.4528903333333325, 296.222094
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
```

```
  Name = Track11
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint10
  CompleteFlag = Atwaypoint11
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -8.166675666666665, 296.222094
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track12
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint11
  CompleteFlag = Atwaypoint12
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.880460999999997, 296.222094
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track13
  TimeOut = 30.0
```

```
  InitialState = OFF
  StartFlag = Atwaypoint12
  CompleteFlag = Atwaypoint13
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -14.492714666666664, 293.3312216666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track14
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint13
  CompleteFlag = Atwaypoint14
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -16.104968333333332, 290.44034933333336
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track15
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint14
```

```
  CompleteFlag = Atwaypoint15
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -17.717222, 287.549477
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track16
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint15
  CompleteFlag = Atwaypoint16
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -18.18419666666667, 272.680669
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track17
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint16
  CompleteFlag = Atwaypoint17
  TimeoutFlag = Stop
```

```
  ErrorFlag = Stop
  Location = -18.651171333333334, 257.811861
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track18
  TimeOut = 30.0
  InitialState = OFF
  StartFlag = Atwaypoint17
  CompleteFlag = Atwaypoint18
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -19.118146, 242.94305300000002
  Radius = 3.0
  Speed = 15.0
}
Task = EndMission
{
  Priority = 1
  Name = AllDone
  TimeOut = Never
  StartFlag = Stop
}
Task = Timer
{
  Priority = 3
  Name = OverallTimeOut
```

```
  TimeOut = 2000.0
  InitialState = On
  TimeoutFlag = Stop
}
```

A.2 Mission Script for Test 2

The mission script generated by Kongming for Test 2 in Section 8.1 is the following.

Name = test2

```
Task = FunctionGenerator
{
    Priority    = 3
    Name        = Z-axis
    TimeOut     = 900
    InitialState    = ON
    ErrorFlag   = EndMission
    FUNCTION    = Constant
    DOF     = HEAVE
    Magnitude   = 15
}
Task = GoToWaypoint
{
    Priority    = 3
    Name        = GoToWaypoint
    TimeOut     = 900
    InitialState    = ON
    CompleteFlag = Atwaypoint0
    TimeoutFlag = Stop
    ErrorFlag   = Stop
    Location   = -18, 240
```

```
    Radius   = 3.0
    Speed    = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track1
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint0
  CompleteFlag = Atwaypoint1
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -17.725779333333332, 254.25188933333334
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track2
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint1
  CompleteFlag = Atwaypoint2
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -17.45155866666666, 268.5037786666667
  Radius = 3.0
  Speed = 15.0
```

```
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track3
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint2
  CompleteFlag = Atwaypoint3
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -17.177338, 282.755668
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track4
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint3
  CompleteFlag = Atwaypoint4
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -3.871832666666661, 291.1529366666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
```

```
{
  Priority = 3
  Name = Track5
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint4
  CompleteFlag = Atwaypoint5
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 9.43367266666667, 299.55020533333334
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track6
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint5
  CompleteFlag = Atwaypoint6
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 22.739178000000003, 307.947474
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
```

```
  Name = Track7
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint6
  CompleteFlag = Atwaypoint7
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 19.017981833333337, 308.3265773333334
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track8
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint7
  CompleteFlag = Atwaypoint8
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 15.296785666666668, 308.7056806666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track9
  TimeOut = 60.0
```

```
  InitialState = OFF
  StartFlag = Atwaypoint8
  CompleteFlag = Atwaypoint9
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 11.575589500000001, 309.084784
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track10
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint9
  CompleteFlag = Atwaypoint10
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 1.8113190000000001, 301.35875400000003
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track11
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint10
```

```
  CompleteFlag = Atwaypoint11
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -7.952951500000001, 293.632724
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track12
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint11
  CompleteFlag = Atwaypoint12
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -17.717222, 285.906694
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track13
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint12
  CompleteFlag = Atwaypoint13
  TimeoutFlag = Stop
```

```
  ErrorFlag = Stop
  Location = -17.927659, 299.89064233333335
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track14
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint13
  CompleteFlag = Atwaypoint14
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -18.138096, 313.8745906666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track15
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint14
  CompleteFlag = Atwaypoint15
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -18.348533, 327.858539
```

```
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track16
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint15
  CompleteFlag = Atwaypoint16
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -16.142481666666665, 327.892744
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track17
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint16
  CompleteFlag = Atwaypoint17
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -13.936430333333334, 327.92694900000004
  Radius = 3.0
  Speed = 15.0
```

```
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track18
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint17
  CompleteFlag = Atwaypoint18
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -11.730379, 327.961154
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track19
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint18
  CompleteFlag = Atwaypoint19
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -11.95832433333333, 313.04798066666666
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
```

```
{
  Priority = 3
  Name = Track20
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint19
  CompleteFlag = Atwaypoint20
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.186269666666664, 298.13480733333336
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track21
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint20
  CompleteFlag = Atwaypoint21
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.414214999999999, 283.221634
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
```

```
  Name = Track22
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint21
  CompleteFlag = Atwaypoint22
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 0.06115566666666794, 271.319273
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track23
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint22
  CompleteFlag = Atwaypoint23
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 12.536526333333335, 259.416912
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track24
  TimeOut = 60.0
```

```
  InitialState = OFF
  StartFlag = Atwaypoint23
  CompleteFlag = Atwaypoint24
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 25.011897, 247.514551
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track25
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint24
  CompleteFlag = Atwaypoint25
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 21.585311666666666, 247.38818333333333
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track26
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint25
```

```
  CompleteFlag = Atwaypoint26
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 18.158726333333334, 247.26181566666668
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track27
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint26
  CompleteFlag = Atwaypoint27
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = 14.732141, 247.135448
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track28
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint27
  CompleteFlag = Atwaypoint28
  TimeoutFlag = Stop
```

```
  ErrorFlag = Stop
  Location = 5.557093333333333, 257.34671933333334
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track29
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint28
  CompleteFlag = Atwaypoint29
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -3.6179543333333317, 267.5579906666667
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track30
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint29
  CompleteFlag = Atwaypoint30
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.793002000000001, 277.769262
```

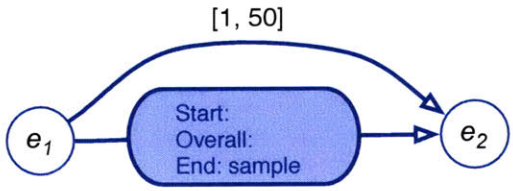
```
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track31
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint30
  CompleteFlag = Atwaypoint31
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.695331999999993, 265.36019466666664
  Radius = 3.0
  Speed = 15.0
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track32
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint31
  CompleteFlag = Atwaypoint32
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.597661999999993, 252.95112733333335
  Radius = 3.0
  Speed = 15.0
```

```
}
Task = GoToWaypoint
{
  Priority = 3
  Name = Track33
  TimeOut = 60.0
  InitialState = OFF
  StartFlag = Atwaypoint32
  CompleteFlag = Atwaypoint33
  TimeoutFlag = Stop
  ErrorFlag = Stop
  Location = -12.499992000000002, 240.54209000000002
  Radius = 3.0
  Speed = 15.0
}
Task = EndMission
{
  Priority = 1
  Name = AllDone
  TimeOut = Never
  StartFlag = Stop
}
Task = Timer
{
  Priority = 3
  Name = OverallTimeOut
  TimeOut = 3000.0
  InitialState = On
  TimeoutFlag = Stop
}
```

Appendix B

B.1 Underwater Scenario 1

Qualitative State Plan (QSP) input:



State variables: posX, posY (x, y position)

Propositional variables: sample

Control variables: velX, velY (x, y velocity)

Δt: 1

Objective: min distance traveled

Initial condition: { (posX, posY) = (0, 0), sample = f }

region(sample): (posX, posY) $\in$ ([80,90], [70,80])

Continuous action type: glide

duration $\in (0, \infty)$

condition-start: { (posX, posY) $\in$ ([0,100], [0,100]) }

condition-overall: { (posX, posY) $\in$ ([0,100], [0,100]) }

condition-end: { (posX, posY) $\in$ ([0,100], [0,100]) }

dynamics: { velX $\in$ [-10, 10], velY $\in$ [-10, 10] }

Discrete action type: take sample

duration $\in$ [2, 8]

condition-start: { (posX, posY) $\in$ region(?sample), ?sample = f }

condition-overall: { (posX, posY) $\in$ region(?sample), ?sample = f }

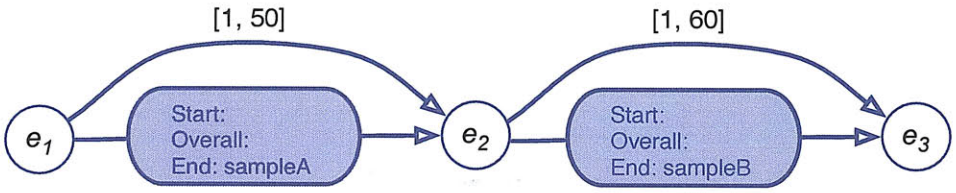
condition-end: { (posX, posY) $\in$ region(?sample), ?sample = f }

effect-end: { ?sample = t }

External constraints: (posX, posY) $\in$ ([0,100], [0,100]) (map boundaries)

B.2 Underwater Scenario 2

Qualitative State Plan (QSP) input:



State variables: posX, posY (x, y position)

Propositional variables: sampleA, sampleB

Control variables: velX, velY (x, y velocity)

Δt: 1

Objective: min distance traveled

Initial condition: { (posX, posY) = (0, 0), sampleA = f, sampleB = f }

region(sampleA): (posX, posY) $\in$ ([25,30], [30,35])

region(sampleB): (posX, posY) $\in$ ([55,60], [40,45])

Continuous action type: glide

duration $\in$ (0, ∞)

condition-start: { (posX, posY) $\in$ ([0,100], [0,100]) }

condition-overall: { (posX, posY) $\in$ ([0,100], [0,100]) }

condition-end: { (posX, posY) $\in$ ([0,100], [0,100]) }

dynamics: { velX $\in$ [-10, 10], velY $\in$ [-10, 10] }

Discrete action type: take sample

duration $\in$ [2, 8]

condition-start: { (posX, posY) ∈ region(?sample), ?sample = f }

condition-overall: { (posX, posY) ∈ region(?sample), ?sample = f }

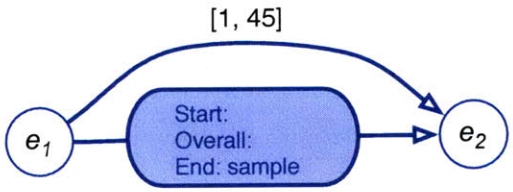
condition-end: { (posX, posY) ∈ region(?sample), ?sample = f }

effect-end: { ?sample = t }

External constraints: (posX, posY) ∈ ([0,100], [0,100]) (map boundaries)

B.3 Underwater Scenario 3

Qualitative State Plan (QSP) input:



State variables: posX, posY, posZ (x, y, z position)

Propositional variables: sample

Control variables: velX, velY, velZ (x, y, z velocity)

Δt: 1

Objective: min distance traveled

Initial condition: { (posX, posY, posZ) = (0, 0, 0), sample = f }

region(sample): (posX, posY, posZ) ∈ ([80,90], [70,80], [30,40])

Continuous action type: glide

duration ∈ (0, ∞)

dynamics: { velX ∈ [-10, 10], velY ∈ [-10, 10], velZ ∈ [0, 0] }

Continuous action type: descend

duration ∈ (0, ∞)

condition-start: { (posX, posY, posZ) ∈ ((-∞, +∞), (-∞, +∞), [0, 40]) }

condition-overall: { (posX, posY, posZ) ∈ ((-∞, +∞), (-∞, +∞), [0, 40]) }

condition-end: { (posX, posY, posZ) ∈ ((-∞, +∞), (-∞, +∞), [0, 40]) }

dynamics: { velX ∈ [-10, 10], velY ∈ [-10, 10], velZ ∈ [1, 5] }

Continuous action type: ascend

duration $\in (0, \infty)$

condition-start: { (posX, posY, posZ) $\in$ ((-∞, +∞), (-∞, +∞), [0, 40]) }

condition-overall: { (posX, posY, posZ) $\in$ ((-∞, +∞), (-∞, +∞), [0, 40]) }

condition-end: { (posX, posY, posZ) $\in$ ((-∞, +∞), (-∞, +∞), [0, 40]) }

dynamics: { velX $\in$ [-10, 10], velY $\in$ [-10, 10], velZ $\in$ [-5, -1] }

Discrete action type: take sample

duration $\in$ [2, 8]

condition-start: { (posX, posY, posZ) $\in$ region(?sample), ?sample = f }

condition-overall: { (posX, posY, posZ) $\in$ region(?sample), ?sample = f }

condition-end: { (posX, posY, posZ) $\in$ region(?sample), ?sample = f }

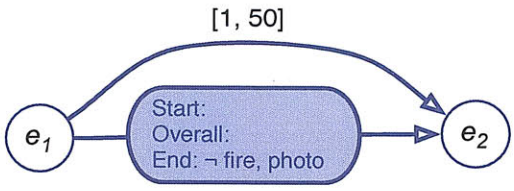
effect-end: { ?sample = t }

External constraints: (posX, posY, posZ) $\in$ ([0,100], [0,100], [0,40]) (within map boundaries)

External constraints: (posX, posY, posZ) $\notin$ ([40,50], [30,40], [30,40]) (outside obstacle)

B.4 Fire-fighting Scenario 1

Qualitative State Plan (QSP) input:



State variables: posX, posY (x, y position)

Propositional variables: fire, water, photo

Control variables: velX, velY (x, y velocity)

Δt: 0.5

Objective: min distance traveled

Initial condition: { (posX, posY) = (0, 0), fire = t, water = f, photo = f }

region(lake): (posX, posY) $\in$ ([15,20], [10,15])

region(fire): (posX, posY) $\in$ ([30,35], [20,25])

Continuous action type: fly

duration $\in (0, \infty)$

dynamics: { velX $\in$ [-2, 2], velY $\in$ [-2, 2] }

Discrete action type: fill water

duration $\in$ [1, 7]

condition-start: { (posX, posY) $\in$ region(?lake), water = f }

condition-overall: { (posX, posY) $\in$ region(?lake), water = f }

condition-end: { (posX, posY) $\in$ region(?lake), water = f }

effect-end: { water = t }

Discrete action type: extinguish fire

duration $\in$ [3, 12]

condition-start: { (posX, posY) $\in$ region(?fire), fire = t, water = t }

condition-overall: { (posX, posY) $\in$ region(?fire), fire = t, water = t }

condition-end: { (posX, posY) $\in$ region(?fire), fire = t, water = t }

effect-end: { fire = f, water = f }

Discrete action type: take photo

duration $\in$ [1, 5]

condition-start: { (posX, posY) $\in$ region(?fire), fire = f, photo = f }

condition-overall: { (posX, posY) $\in$ region(?fire), fire = f, photo = f }

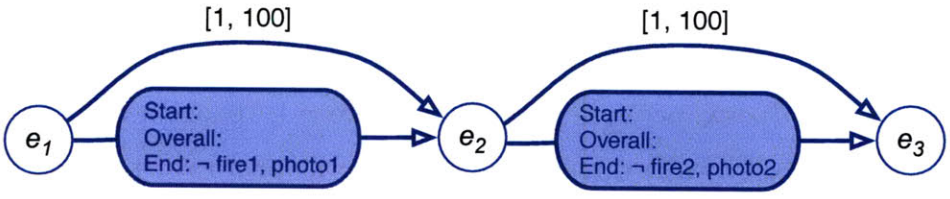
condition-end: { (posX, posY) $\in$ region(?fire), fire = f, photo = f }

effect-end: { photo = t }

External constraints: (posX, posY) $\in$ ([0,100], [0,100]) (map boundaries)

B.5 Fire-fighting Scenario 2

Qualitative State Plan (QSP) input:



State variables: posX, posY (x, y position)

Propositional variables: fire1, water, photo1, fire2, photo2

Control variables: velX, velY (x, y velocity)

Δt: 0.5

Objective: min distance traveled

Initial condition: { (posX, posY) = (0, 0), fire1 = t, water = f, photo1 = f, fire2 = t, photo2 = f }

region(lake): (posX, posY) $\in$ ([20,25], [30,35])

region(fire1): (posX, posY) $\in$ ([40,45], [45,50])

region(fire2): (posX, posY) $\in$ ([70,75], [55,60])

Continuous action type: fly

duration $\in$ (0, ∞)

dynamics: { velX $\in$ [-15, 15], velY $\in$ [-15, 15] }

Discrete action type: fill water

duration $\in$ [1, 7]

condition-start: { (posX, posY) $\in$ region(?lake), water = f }

condition-overall: { (posX, posY) $\in$ region(?lake), water = f }

condition-end: { (posX, posY) $\in$ region(?lake), water = f }

effect-end: { water = t }

Discrete action type: extinguish fire

duration $\in$ [3, 12]

condition-start: { (posX, posY) $\in$ region(?fire), ?fire = t, water = t }

condition-overall: { (posX, posY) $\in$ region(?fire), ?fire = t, water = t }

condition-end: { (posX, posY) $\in$ region(?fire), ?fire = t, water = t }

effect-end: { ?fire = f, water = f }

Discrete action type: take photo

duration $\in$ [1, 5]

condition-start: { (posX, posY) $\in$ region(?fire), ?fire = f, ?photo = f }

condition-overall: { (posX, posY) $\in$ region(?fire), ?fire = f, ?photo = f }

condition-end: { (posX, posY) $\in$ region(?fire), ?fire = f, ?photo = f }

effect-end: { ?photo = t }

Bibliography

[ABS97] D. Avis, D. Bremner, and R. Seidel. How good are convex hull algorithms? *Comput. Geometry: Theory Appl., vol. 7, pp. 265301*, 1997.

[AFH96] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 1996.

[AHH96] R. Alur, T. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. on Software Engineering, 22(3):181-201*, 1996.

[ASW98] C. Anderson, D. Smith, and D. Weld. Conditional effects in graphplan. *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 1998.

[auv] `http://auvlab.mit.edu`.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Proceedings of TACAS*, 1999.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 1997.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results. *Proceedings of ECP-99*, 1999.

[BG01] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[BK96] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Proceedings of AAAI*, 1996.

[BM06] J. Baier and S. McIlraith. Planning with first-order temporal extended goals using heuristic search. *Proceedings of AAAI*, 2006.

[BT97] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

[CCFL09] A. J. Coles, A. I. Coles, M. Fox, and D. Long. Temporal planning in domains with linear processes. *Proceedings of International Joint Conference on Artificial Intelligence*, 2009.

[CFLS08a] A. Coles, M. Fox, D. Long, and A. Smith. A hybrid relaxed planning graph-lp heuristic for numeric planning domains. *Proceedings of ICAPS*, 2008.

[CFLS08b] A. Coles, M. Fox, D. Long, and A. Smith. Planning with problems requiring temporal coordination. *Proceedings of AAAI*, 2008.

[Chi97] J. Chinneck. Feasibility and viability. *Advances in Sensitivity Analysis and Parametric Programming, International Series in Operations Research and Management Science*, 1997.

[Chu08] S. Chung. Model-based planning through constraint and causal order decomposition. *PhD Thesis, MIT*, 2008.

[CK98a] A. Chutinan and B. Krogh. Computing approximating automata for a class of linear hybrid systems. *Hybrid Systems V, Lecture Notes in Computer Science*, 1998.

[CK98b] A. Chutinan and B. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. *Proceedings of the 37rd IEEE Conference on Decision and Control*, 1998.

[DK01a] M. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Artificial Intelligence Journal 132 (p.151-182)*, 2001.

[DK01b] M. Do and S. Kambhampati. Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of ECP*, 2001.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Journal of Artificial Intelligence, 49 (1991) 61-95*, 1991.

[dV93] J. Van de Vegte. Feedback control systems. *Prentice Hall, 3rd Edition*, 1993.

[FL01] M. Fox and D. Long. PDDL+ Level 5: An Extension to PDDL2.1 for Modelling Planning Domains Continuous Time-dependent Effects. *Available at http://www.dur.ac.uk/d.p.long/competition.html*, 2001.

[FL03] M. Fox and D. Long. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 2003.

[FN71] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2:189-208*, 1971.

[Gir05] A. Girard. Reachability of uncertain linear systems using zonotopes. *Hybrid Systems: Computation and Control, ser. Lecture Notes in Computer Science, vol. 1790, pp. 3145*, 2005.

[GS02] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, 2002.

[GSS04] A. Gerevini, A. Saetti, and I. Serina. Planning with numerical expressions in LPG. *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, 2004.

[HG01] P. Haslum and H. Geffner. Heuristic planning with time and resources. *Proceedings of European Conference on Planning (ECP'01)*, 2001.

[HN01] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14, 253-302*, 2001.

[HO99] J. Hooker and M. Osorio. Mixed logical-linear programming. *Discrete Applied Mathematics 96-97, 395-442*, 1999.

[Hof03] J. Hoffmann. The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 2003.

[Hof06] A. Hofmann. Robust execution of bipedal walking tasks from biomechanical principles. *PhD Thesis, MIT*, 2006.

[HW06] A. Hofmann and B. Williams. Robust execution of temporally flexible plans for bipedal walking devices. *Proceedings of ICAPS*, 2006.

[Ins] Monterey Bay Aquarium Research Institute. http://www.mbari.org/.

[KD00] J. Kvarnström and P. Doherty. TALPlanner: A temporal logic based forward chaining planner. *Annals of Mathematics Artificial Intelligence*, 2000.

[KGBM04] M. Kvasnica, P. Grieder, M. Baotić, and M. Morari. Multi-parametric toolbox (mpt). *Hybrid Systems: Computation and Control, ser. Lecture Notes in Computer Science, vol. 2993, pp. 448462*, 2004.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. *Proceedings of the 4th European Conference on Planning (ECP'97)*, 1997.

[Kos01] E. Kostousova. Control synthesis via parallelotopes: Optimization and parallel computations. *Optim. Meth. Software, vol. 14, no. 4, pp. 267310*, 2001.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. *Proceedings of European Conference on Artificial Intelligence*, 1992.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. *Proceedings of IJCAI*, 1999.

[KV07] A. Kurzhanskiy and P. Varaiya. Ellipsoidal techniques for reachability analysis of discrete-time linear systems. *IEEE Transactions on Automatic Control, Vol. 52, NO. 1,*, 2007.

[LBHJ04] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple bounded LTL model checking. *Proceedings of FMCAD*, 2004.

[Léa05] T. Léauté. Coordinating agile systems through the model-based execution of temporal plans. *Master Thesis, MIT*, 2005.

[LF99] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research 10, 87-115*, 1999.

[LF02] D. Long and M. Fox. Fast temporal planning in a graphplan framework. *Proceedings of International Conference on Automated Planning and Scheduling*, 2002.

[LW05a] T. Léauté and B. Williams. Coordinating agile systems through the model-based execution of temporal plans. *Proceedings of AAAI*, 2005.

[LW05b] H. Li and B. Williams. Generalized conflict learning for hybrid discrete linear optimization. *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 2005.

[MBB+09] N. Meuleau, E. Benazera, R. Brafman, E. Hansen, and Mausam. A heuristic search approach to planning with continuous resources in stochastic domains. *Journal of Artificial Intelligence Research*, 2009.

[McD96] D. McDermott. A heuristic estimator for means-ends analysis in planning. *Proceedings AIPS-96*, 1996.

[McD03] D. McDermott. Reasoning about autonomous processes in an estimated-regression planner. *Proceedings of ICAPS*, 2003.

[MR07] R. Mattmuller and J. Rintanen. Planning for temporally extended goals as propositional satisfiability. *Proceedings of International Joint Conference on Artificial Intelligence*, 2007.

[MtAPCC98] D. McDermott and the AIPS '98 Planning Competition Committee. PDDL - the planning domain definition languages. *Technical Report, Available at: http://www.cs.yale.edu/homes/dvm*, 1998.

[ody] `http://seagrant.mit.edu/auvwiki/index.php/Main_Page`.

[Pea84] J. Pearl. Heuristics: Intelligent search strategies for computer problem solving. *Addison-Wesley*, 1984.

[PR96] M. Parker and J. Ryan. Finding the minimum weight IIS cover of an infeasible system of linear inequalities. *Annals of Mathematics and Artificial Intelligence 17*, 1996.

[PW92] S. Penberthy and D. Weld. UCPOP: A sound, complete, partial-order planner for ADL. *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92)*, 1992.

[PW94] S. Penberthy and D. Weld. Temporal planning with continuous change. *Proceedings of AAAI*, 1994.

[rep] `http://people.csail.mit.edu/rootless/replanning.html`.

[SD05] J. Shin and E. Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166, 2005.

[SK03] O. Stursberg and B. Krogh. Efficient representation and computation of reachable sets for hybrid systems. *Hybrid Systems: Computation and Control, ser. Lecture Notes in Computer Science, vol. 2623, pp. 482497*, 2003.

[SW99] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI*, 1999.

[Ted09] R. Tedrake. LQR-trees: Feedback motion planning on sparse randomized trees. *Proceedings of Robotics: Science and Systems (RSS)*, 2009.

[who] `http://www.whoi.edu`.

[WICE03] B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Invited Paper, Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 2003.

[WN97] B. Williams and P. Nayak. A reactive planner for a model-based executive. *Proceedings of IJCAI*, 1997.

[WW99] S. Wolfman and D. Weld. The LPSAT engine and its application to resource planning. In *Proceedings of IJCAI*, 1999.

[Zha92] F. Zhao. Automatic analysis and synthesis of controllers for dynamical systems based on phase-space knowledge. *PhD Thesis, MIT*, 1992.