Otto-Friedrich-Universität Bamberg

Professur für Angewandte Informatik, insbesondere Smart Environments

Masterarbeit

im Studiengang Angewandte Informatik
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Zum Thema:

Temporal HTN Planning with a Stream of Tasks: Anticipating Interleavability for Replanning

Vorgelegt von:

Felix Haase

Themensteller:
Prof. Dr. Diedrich Wolter

Abgabedatum:
11.04.2024

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

Abstract

1 Introduction

2 Related Work

2.1 HTN Planning

Hierarchical Task Network (HTN) planning is a planning paradigm that allows for the specification of complex tasks by decomposing them into the subtasks. The planning is finished when there is no task left that has subtasks, the result is also called a Decomposition Tree. HTN Planning can be seen as a guided form of traditional planning, since the task structure is given. HTN Planning problems can be defined in several languages, such as the SHOP syntax, supported by SHOP-Like planners, ANML Language - supported by the FAPE planner and HDDL Syntax which is expected to become the standard. While these languages mostly support the same features, it is important to note that HDDL currently only supports a small subset of the PDDL syntax it is based on, primarily because the most HTN planners do not support temporal or resource planning. The HDDL 2.1 proposal adds the support for Temporal planning, but there is currently no public system that supports this syntax. The most expressive Syntax is the ANML-Language, which also supports dynamic fluents, that change over time. This Syntax is used throughout the whole thesis, as it supports Time for HTN Planning and is much less verbose than SHOP. The Fape planner is therefore used as a basis for this work.

2.1.1 ANML Language

The ANML Language is has a similar syntax very similar to common Programming languages like Java or C++, instead of being inspired by lisp syntax like PDDL or SHOP.

Types:

It is possible to define Types, subtypes and optionally include constants or fluents on these types.

```
type Location;

type NavLocation < Location;

type Robot < Location with {
  variable NavLocation location;
};

type Item with {
  variable Location location;
};
```

Functions:

Functions can also be defined outside of a type definition and variable and fluent are aliases for a function. They take may take any number of parameters and have a return type. Predicate is an alias for a function with a boolean return type. Functions are defined for a specific time range and may change their value over time.

```
variable Room houseCenter; // is the same as
function Room houseCenter();
```

```
function boolean connected(Location a, Location b);
predicate connected(Location a, Location b);
```

It is also possible to create constant functions, which are defined using the constant keyword. These values do not change over time and need to be set in the problem definition.

```
constant boolean connected(Location a, Location b);
```

Logical Statements:

A logical statement describes changes and conditions on state variables. It is associated with a start and en time point. They refer to state variables: ANML functions with parameters.

1. **Persistence:** `connected(Kitchen, Entrance)==true;` requires the state variable `connected (Kitchen, Entrance)` to be **true** between the start and end time points of the statement.
2. **Assignment:** `connected(Kitchen, Entrance):=true;` specifies that the state variable `connected(Kitchen, Entrance)` will have the value **true** at the end of the statement and is undefined between start and end.
3. **Transition:** `connected(Kitchen, Entrance)==false :->true` requires the state variable to have the value false at the start of the statement and specifies that it will have value **false** at the end of the statement.

Temporal annotations:

Temporal annotations are temporal constraints on the timepoints of statements. Given a statement `s`, where **start**`(s)` and **end**`(s)` represent its start and end time-points.

```
[all] s; or [start,end] s;
=> start(s) == start && end(s) == end
[start+10, end-5] s;
=> start(s) == start +10 && end(s) == end-5
[2, 10] s;
=> start(s) == 2 && end(s) == 10;
// or any combination of the above annotations
```

In the preceding text, **start** and **end** refer to the start and end time points of the interval containing the annotated statement (such as an action or a problem).

It is possible to give the same annotation to several statements:

```
[start, end] {
  connected(a, b) == true;
  r.canGo(a) := false;
};
// is equivalent to
[start, end] connected(a, b) == true;
[start, end] r.canGo(a) := false;
```

A temporal annotation with the ‘contains‘ keyword means that the given statement must be included in the interval:

```
[start , end] contains s;
// start(s) >= start && end(s) <= end
```

WARNING: in its current implementation, the **contains** keyword differs from the mainline ANML definition since it does *not* require the condition to be false before and after the interval. In fact, the above statement is equivalent to `[start ,end] contains [0] s;` (but this notation is not supported in FAPE)

Action are operators having typed parameters and that might contain any number of statements. In the following example, the transition statement's start and end timepoints are equals to those of the action.

```
action Move(Robot r, Location a, Location b) {
  [start , end] {
    r.location == a :-> b;
  };
};
```

Actions can be given a duration either fixed or parameterized with an invariant function of type integer.

```
constant integer travel_time(Loc a, Loc b);
travel_time(A, B) := 10;
travel_time(A, C) := 15;

action Move2(Robot r, Loc a, Loc b) {
  duration := travel_time(a, b);
  //...
};
```

The duration can also be left uncertain in a given interval using the keyword ‘:in‘.

```
action Move3(Robot, Loc a, Loc b) {
  duration :in [10, 15];
  //...
};

constant integer min_travel_time(Loc a, Loc b);
min_travel_time(A, B) := 10;
min_travel_time(A, C) := 15;
constant integer max_travel_time(Loc a, Loc b);
max_travel_time(A, B) := 13;
max_travel_time(A, C) := 18;

action Move4(Robot r, Loc a, Loc b) {
  duration :in [min_travel_time(a,b), max_travel_time(a,b)];
};
```

If this notation is used, a contingent constraint will be considered between the start and end time points of the action. It is handled through an STNU framework.

Actions can be associated with a set of decomposition, if it has one or more decomposition,

we call it a non-primitive action. In a valid plan, every action must be associated with exactly one of its decomposition. The choice of this decomposition is branching point in the search.

There is no restriction to what can be written in decomposition. Hence it can be used to represent:

- methods in an HTN fashion - conditional effects - a controllable choice over the effects of an action. We discourage this practice, since it will not be apparent in the plan.

The following example shows subtasks in decompositions. Those are described in the next sub-section.

```
action Pick(Robot r, Item i) {
  :decomposition{
    [all] PickWithLeftGripper(r, i);
  };
  :decomposition{
    [all] PickWithRightGripper(r, i);
  };
};

action Go(Vehicle v, Loc from, Loc to) {
  [start] location(v) == from;
  :decomposition{
      isCar(v) == true;
      [all] GoByRoad(v, from, to);
  };
  :decomposition{
    isPlane(v) == true;
    [all] Fly(v, from, to);
  };
};
```

The last example showed a usage of actions as condition appearing in the decomposition of another one. This condition, called task, is satisfied if there is an action with the same name and parameters satisfying all temporal constraints on the action condition time points.

```
[10,90] Go(PR2, Kitchen, Bedroom);
```

The above condition will be satisfied iff there is an action ‘Go‘ with the parameters ‘(PR2, Kitchen, Bedroom)‘ that starts exactly at 10 and ends exactly at 90.

WARNING: this differs from the mainline ANML definition because the considered time-points for temporal constraints are those of the action itself.

```
// this will be satisfied if there is an Go with this
// parameters that starts and end within the
// interval [10,90]
[10, 90] contains Go(PR2, Kitchen, Bedroom;

// this action must have exactly the same duration as the
// one of the ConcreteGo action on which it is conditioned
action AbstractGo(Loc a, Loc b) {
  [all] ContreteGo(a, b);
```

```
};

action ConcreteGo(Loc a, Loc b) {
  duration :in [min_travel_time(a,b), max_travel_time(a,b)];
};
```

Temporal constraints can be specified between intervals (i.e. any ANML object with start and end time-points such as actions or statements). The interval must be given a local ID:

```
  [all] contains {
      idA : I.location == A;
      idB : I.location == B;
  };

  // specifies that the second statement must start at least
     10 times units
  // after the end of the first one
  end(idA) +10 < start(idB);

  // specifies that the second statement must end exactly 60
     time units before
  // the end of the containing interval (i.e. the action if
     the statement is defined
  // in an action or the plan if the statement is defined in
     the problem).
  end(idB) = end -60;
```

It is also possible to put temporal constraints between actions conditions. The ‘ordered‘ and ‘unordered‘ keywords are not supported but can be replaced by temporal constraints as in the following example.

```
  action PickAndPlace(Robot r, Item i) {
    pickID : Pick(r, i);
    placeID : Place(r, i);
    end(pickID) < start(placeID);
  };
```

Constraints can be expressed between variables and invariant functions.

```
  constant Country country_of(City c);
  instance Contry France, US, Germany;

  country_of(Paris) == country_of(Toulouse);
  country_of(Chicago) != country_of(Paris);

  // creates a variable "a_country" and constrains it
  // to be different to Paris' country
  constant Country a_country;
  country_of(Paris) != a_country;
```

Resources are not supported yet. An initial implementation is currently in the source repository but is not stable enough for daily usage.

The FAPE planner is a temporal planner reasoning in plan space. As such, it mainly reasons with flaws/resolvers to reach a solution plan. As long as there is no task conditions in the domain, FAPE will act as a plan space planner and will try to solve the current open goals and threats in its plan.

It uses a lifted representation and timelines as a time oriented view of the evolution of state variables.

Temporal constraints are managed in an STN extended for the integration of contingent constraints (e.g. the uncertain duration of an action). This STNU framework can be used, while planning different types of consistency: (i) STN consistency (ii) pseudo-controllability (iii) dynamic-controllability.

A binding constraint manager is used to enforce the equality and difference constraints between variables (e.g. parameters of actions).

To put things short: without task conditions, FAPE is a lifted temporal planner, searching in plan-space.

In addition to generative planning problems, ANML allows the definition of hierachical problems. THis is done through the definition of of task that should be fulfilled. Such task can appear

- in the problem statement. In which case they are goal tasks (usually refered as the initial task network in the HTN literature) - in actions. In which case they are subtasks of this actions.

Any task appearing in the plan (either as part of the problem definition or inserted with an action) must be refined. We say that a task ‘[t1,t2] name(a1...an)‘ is refined if there is an action ‘name(a1,...an)‘ that starts at ‘t1‘ and ends at ‘t2‘.

The open goal and threats flaws from plan-space planning are completed by an *unrefined task* flaw associated to each task that has not been refined yet.

A (very) simplified view of FAPE’s search procedure is:

1) refine all tasks. Starting from the initial task network insert all actions necessary for all tasks to be refined. If any subtasks are inserted during this process, recursively refine those as well. 2) Make sure the plan is consistent by handling all open goals and threats. This implies enforcing causal constraints (all actions’ conditions must be supported) and temporal consistency (their should not be two concurrent and conflicting activities). Enforcing causal constraints generally requires the introduction of new actions in the partial plan.

Actions might be inserted as standalone resolvers (e.g. to solve an open goal flaw). This differs from the HTN way where any action in the plan is derived from the initial task network. To mimic this specificity, ANML provides the ANML keyword: a ‘motivated‘ action needs to refine a task (i.e. it must be part of some hierarchy). In practice, it means that only non-motivated actions can be inserted outside of the action hierarchy. Making all actions motivated and giving a root action will result in a HTN-like search, expending a search tree from the initial task network only.

Unless really sure of what you do, you should avoid having both motivated actions and

goals expressed as statements over state variable. The intended way of using FAPE in this setting is:

- with one or more goal tasks, forming the initial task network - motivated actions should be derivable from those goal tasks and will be the skeleton of the plan. - non-motivated action are used to handle corner-cases that were not described in the task network.

The following example shows a very simple hierarchical domain where the skeleton of the plan will derive from Transport. Note that Pick and Drop are motivated and hence can not appear outside of Transport. However the Move action will be freely inserted in the plan to tackle open-goals flaws on the location of the robot.

A typical planning process for this problem would be the following:

3 Methodology

4 Experimental Evaluation

5 Discussion

6 Conclusion

7 Acknowledgements

First of all, I want to thank Prof. Dr. Diedrich Wolter, my supervisor, for the opportunity to work on this topic for my thesis. He helped me with the brainstorming of possible approaches and referred me to different topics of interest.

Further thanks go to my fellow students, friends, and family for the ongoing support during the time of writing and coding. Special thanks to Tobias Schwartz, Mai Thanh Ha Dinh, Jens Dümler, and Bernhard Walter Freiherr von Rotenhan for proofreading and feedback on the thesis.

References

Ich erkläre hiermit gemäß §17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bamberg, den 30.07.2020 Felix Haase