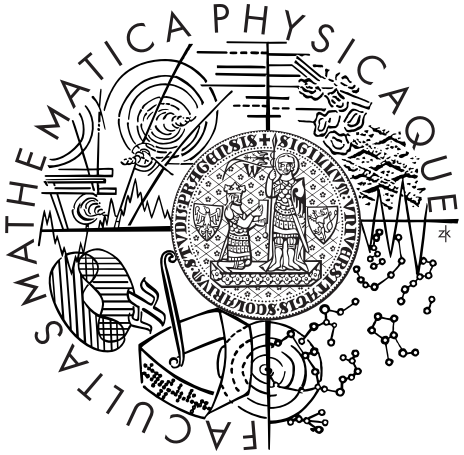
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Mgr. Filip Dvořák

Integrating Planning and Scheduling

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the doctoral thesis: prof. RNDr. Roman Barták, Ph.D.

Study program: Theoretical Computer Science

Specialization: AI Planning

Prague 2014

I am deeply grateful to Roman Barták for his continual support and supervision during my doctoral studies, and for his constructive criticism that helped to make this thesis better.

I would like to deeply thank Malik Ghallab and Felix Ingrand for sharing their expert opinions on planning and robotics, providing helpful criticism, and for the opportunity to join them in their research efforts.

I am deeply grateful to Daniel Toropila and Arthur Bit-Monnot for joining forces on our research paths, contributions to implementation and experimental setups.

All faults in this thesis, if any, are, of course, mine.

I cannot thank enough my parents for their continual support and encouragement during my studies.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Integrace plánování a rozvrhování

Autor: Mgr. Filip Dvořák

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí disertační práce: prof. RNDr. Roman Barták, Ph.D., KTIML MFF UK

Abstrakt: Hlavním tématem práce je navržení a vývoj plánovacího systému FAPE, který integruje explicitní reprezentaci času, rozhodování s diskrétními zdroji a rezervoáry, a hierarchické dekompozice. FAPE je první plánovací systém, který akceptuje jazyk ANML a podporuje většinu jeho hlavních vlastností. V práci se zabýváme různými aspekty integrace zmíněných konceptů, také navrhneme novou techniku pro reformulaci plánovacích problémů do reprezentace pomocí stavových proměnných a identifikujeme přechod výkonosti mezi používaním řídkých a minimálních časových sítí. FAPE dále rozšíříme o schopnosti konání a experimentálně vyhodnotíme výkonnost a výhody jeho vysoké expresivity. Na závěr předvedeme FAPE jako systém, který umí plánovat i konat v experimentech v realném světě, kdy FAPE ovládá robota PR2.

Klíčová slova: plánování, rozvrhování, temporální podmínky, HTN, robotika

Title: Integrating Planning and Scheduling

Author: Mgr. Filip Dvořák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., KTIML MFF UK

Abstract: The main topic of the work is the design and development of a plan-space planning system FAPE that integrates explicit time reasoning, resource reasoning with discrete resources and reservoirs and hierarchical decompositions. FAPE is the first planning system that accepts the language ANML, supporting most of its major features. We investigate different aspects of the integration, also proposing a new problem reformulation technique for the state-variable representation and discovering a transition of performance between sparse and minimal temporal networks. We further extend FAPE with acting capabilities and evaluate the runtime properties and benefits of its expressiveness. Finally, we present FAPE as a planning and acting system in real world experiments, where FAPE operates a PR2 robot.

Keywords: planning, scheduling, temporal constraints, HTN, robotics

Contents

Introduction

Planning is a process of reasoning how to effect the world through actions towards some desired state of the world. It is an art of thinking before acting and as such the automation of planning has been considered one of the key elements of the Artificial Intelligence since its beginning.

While Planning is expected to find a set of actions to achieve the goal, Scheduling is focused on reasoning when the actions should be executed to satisfy some resource constraints. Planning problems may also contain resources, however, the extension towards the scope and efficiency of handling resources in scheduling has been a long standing challenge giving the motivation and main focus of this thesis.

Planning requires an ability to predict the result of future changes that can be produced in the system. To predict the effects of changes we need to formulate a model of a dynamic system; in planning terminology the "model of the world". One of the earliest efforts to construct an automated reasoning system is known as the classical STRIPS planning (Fikes and Nilsson, 1972), which was domain-independent approach strongly inspired by predicate logic. We can commonly find less general planning models used in particular planning fields such as motion planning (Hwang and Ahuja, 1992), industrial planning (Aylett et al., 2000) and rescue planning (Biundo and Schattenberg, 2001) to name a few, but for the purpose of this thesis we focus on *domain-independent planning*, which is the key feature spread across modern planning systems - the planner provides an expressive language and planning capabilities for any problem specified in the language. Comparing to the development of domain-specific solvers, the domain-independent planning offers faster integration, adaptability and competitive performance, supported by the reusability of large number of domain-independent techniques. However, under the assumption of deterministic observable system (the results of actions are well defined and we have a complete information about the world) with a finite number of objects, finding the plan is already a PSPACE-hard problem (Erol et al., 1995b). As a result, many approaches to planning are based on heuristic search techniques using a wide variety of subroutines from other fields such as graph theory (Muscettola, 2002) or integer programming (Coles et al., 2014). Some approaches just transform the planning problem into another problem, constraint satisfaction (Beek and Chen, 1999) and satisfiability (Kautz and Selman, 1992) being the most common. Another approach how we deal with the high complexity of planning is through Hierarchical Task Networks (HTN) (Sacerdoti, 1975). It leverages the idea that often the domain designer can add further information about the problem by encoding the way how the sets of actions can be composed into higher level actions, creating a hierarchy of *tasks*; as such, HTN planning has been one of the most widely applied planning approaches in practice (Ghallab et al., 2004). We shall introduce the main concepts of planning in Chapter 1.

Planning with Time and Resources Constraints

To start reasoning with resources in planning, we need to introduce the concept of time into planning. While in planning without notion of time, the plans are sequences of actions, in temporal planning the plans become *schedules* describing when each action starts and ends. We can find different capabilities of reasoning with time within planning. Most contributions to temporal planning abstract away time as *state transitions*, considering only the start and the end of each action on the basis of (Fox and Long, 2003). The motivation comes mainly from the wealth of search techniques and domain independent heuristics that have been developed for classical planning in recent decades. However explicit time is required in many applications, e.g. when dealing with synchronization of action effects, managing deadlines on actions, handling concurrency of actions and reasoning with resources. The *timeline representation*, in systems such as $I_X T_E T$(Laborie and Ghallab, 1995a), instead captures the whole evolution of the facts in the world, allowing to refer to instants beyond starting and ending points of actions. The quantified temporal relations between instants of the timelines are further handled by the *temporal networks* (Dechter et al., 1991). We shall focus on planning with time in Chapter 2.

In planning we deal with objects and their properties. In some situation, we are not interested in the individual objects, but how many are available that share some property or belong to a category, e.g., renting a bike in a citybike rack. The same object can appear as a resource for some action and as an individual object for another one, e.g., once we pick up one bike, this bike is individualized for our remaining use. Sometimes the use is temporary (renting) sometimes it is permanent (buying the bike). The resource can be seen as an abstraction of uniqueness (aggregating several homogeneous entities into a single numerical entity, representing them together), a natural operation people perform when dealing with quantifiable entities in the world, starting with counting money or apples in a basket. The complementary part of abstracting away the uniqueness is the ability to make relative changes upon a resource entity - a passenger occupying one seat represents a consumption of space that is independent on other consumptions as long as the capacity is sufficient. We further investigate the role of resource in planning in Chapter 3.

Contributions

The thesis puts together work done in a span of several years and some parts have been previously published by the author, cooperating and co-authoring Roman Barták, Malik Ghallab, Daniel Toropila, Félix Ingrand and Arthur Bit-Monnot.

The main result of this thesis is the design and development of a new planning system with time and resource constraints *FAPE* (Flexible Acting, Planning and Execution). The contribution lies first in development of a new technique for an automated domain reformulation into a finite-domain state-variable representation described in Section 1.5. Exhaustive experimental evaluation shows that the translation technique produces problems that lead to improved performance of some planners and domains while staying comparatively fast to competing

approaches. Second contribution, covered in chapters 4 and 5, is the unique combination of features provided by *FAPE*. Being a first planner to support Action Notation Modeling Language (ANML) (Smith et al., 2008), *FAPE* seamlessly integrates plan-space planning, HTN and resource reasoning into one system. The last contribution is enriching the planning capabilities of $FAPE$ with acting, allowing *FAPE* to provide a real-time control for any system that can specify its behavior through ANML.

Organization

In this thesis we first look into the classical planning and its flavors, problem representation and common approaches for solving it in Chapter 1. Then we extend planning with explicit time and focus on temporal networks in Chapter 2. We follow up by investigating the concept of resources in planning through the Chapter 3. Chapter 4 focuses on the description of the proposed planning system, where we discuss in detail the design choices. The last chapter of the thesis presents the experiments performed with *FAPE*, both by testing the performance in simulations and allowing *FAPE* to control a robot in real environment.

1. Planning

The classical planning views the planning problem as a *state-transition problem* (Green, 1969). The world is seen as a space of states, where a single state represents a snapshot of the world and the actions represent the transitions between different states. Then the goal of the planner is to find a path through the state space from the initial state to a goal state. We define the *state-transition system* as $\sum = (S, A, \xi)$, where S is a set of states, A is a set of actions and $\xi : S \times A \to S$ is a transition function. We further assume that the state space is finite, the actions transform the state in a deterministic way and the world is static (no external events may change the world state). The propositional STRIPS representation (Fikes and Nilsson, 1972) describes the world through a set atoms (facts) V about the world (such as "the dog is in the house") and further defines the state of the world $s \in S$ as an assignment of truth values to V, we conveniently represent the state of the world s as a set of atoms that hold true and assume the others are false. An action $a \in A$ is then formed from a set of preconditions *pre*, set of positive effects *eff* and a set of negative effects *del*. The preconditions *pre* is a set of atoms that must hold true before the action can be applied, the set of positive effects eff are those atoms that become true by the action application, and *del* is the set of atoms that become false by the action application. The transition function is then defined as follows:

$$\forall a \in A, \forall s \in S, pre(a) \in s, \xi_{strips}(a, s) = (s \cup eff(a)) \setminus del(a)$$

We can now define the planning problem in the classical representation:

Definition 1.0.1. *For a state transition system* $\sum = (S, A, \xi_{strips})$*, classical planning problem is a quadruple (V,I,G,A), where*

- *V is a set of atoms.*
- *I is a set of atoms that are true at the initial state, where the initial state is denoted as* s_0*.*
- *G is a set of atoms that shall be true in the goal state.*
- *A is a set of actions, where each action is a triple (pre,eff,del).*

The solution of the planning problem (V,I,G,A) is a sequence of actions $\pi = (a_1, ..., a_n)$ *such that*

$$\xi_{strips}(\xi_{strips}(\xi_{strips}(s_0, a_1), ...), a_n) = s_g \text{ and } G \subseteq s_g.$$

We call π *a plan for the problem (V,I,G,A).*

Note that while I specifies exactly which atoms are true and false at the beginning, the goal description G is partial in a sense that we only care about achieving the specific set of atoms where the other may, or may not be true.

Since the planning problem is generally hard (Erol et al., 1995a), the algorithmic dominant approach today is the search enhanced by heuristics, which we shall investigate in Section 1.1. Finding the plan satisfies the planning problem,

however we are often further interested in the quality of the produced plan. To evaluate the plan quality in a domain we define metrics such the total number of actions in the plan, the sum of all costs of actions in the plan or the total time needed to execute the plan (makespan). Based on how we handle the plan quality we can find the following flavors of planning:

- Optimal Planning focuses on finding and proving that the quality of the produced plan is the best achievable. The optimality property of plans has a significant meaning in certain problems, e.g. when constructing a business process or game scenario (Pizzi et al., 2010) the optimality guarantees that we have not missed any loop-hole. Optimal planning stays to be a strong research direction.

- Satisfaction Planning is concerned only with finding any plan. Often the space of all plans is very rich (infinite) and we can find the first plan quickly, although such plan may be very inefficient. In the recent decade, the Satisfaction Planning is slowly being abandoned by the research community in favor of the next approach.

- Satisficing Planning is an approach that tries to produce *satisfying* plans that are of a *sufficiently* good quality. The approach can be seen as an application of the best-effort principle, where we try to produce the best we can during the given time - starting from the first found plan and continually improving until the optimal plan is reached or we have run out of time. The approach became quite popular in the recent decade mainly through appearance at the International Planning Competition (Olaya et al., 2011).

- Planning with Preferences adds further details on how we can specify the quality of the plan. We allow the domain designer to provide soft constraints and goals for the problem and the quality of the plan is then evaluated with regard to them. Using our toy example a soft goal can be a requirement that the robot ends up at certain location, while the constraint can be that the robot uses only one gripper at all times. As the requirements coming from the real world problems are getting more complex, this approach is becoming more popular (Edelkamp and Kissmann, 2008).

Most of the classical planning systems today plan through search guided by heuristics. Planners whose search is complete are also naturally suited for optimal planning, e.g. (Domshlak et al., 2011b), (Domshlak et al., 2011a), (Helmert et al., 2011) and (Richter and Westphal, 2008) and they often provide a switch between optimal and satisficing planning. An important and strong field of planning we shall not further investigate in this thesis is planning and reasoning under uncertainty (Collins and Pryor, 1995).

In the following section we shall look into the search techniques for planning. Then we expand on the concepts of hierarchies in planning and finally, we investigate an alternative state-variable representation for planning.

1.1 Search

Searching for a plan is the most common approach for solving a planning problem algorithmically. Having a formal model given in Definition 1.0.1, we can start searching through the space of states, an example of straight-forward plan search is the Algorithm 1. The algorithm exhaustively explores the search space of all possible plans in a forward fashion, starting from the initial state, and returns the first plan found. It is complete and guided by a heuristic for ordering of actions at line 6, it may turn to be quite efficient. An alternative from progressing from the initial state towards the goal is to start from the goal and regress towards the initial state, we call such approach *backward-planning*. The main advantage of the algorithm is its memory efficiency, since it is linearly proportional to the length of the plan, and several decades ago, when a similar algorithm has been proposed for the original STRIPS model (Fikes and Nilsson, 1972), it has been a strong motivation. The obvious disadvantage is that the search can get lost deep in a unpromising branch, while the actual plan is short; there are several techniques to avoid the behavior, such as iterative deepening (continually increasing the length of the plan we search for) or discrepancy search (Harvey and Ginsberg, 1995). Further, the state space we are exploring is quite large, $O(2^n)$ where n is the number of atoms, and consequently the theoretical maximal depth of the search is $O(2^n)$ as well.

Algorithm 1 Forward-Search DFS

Input: (A, s_0, G), where A is a set of actions, s_0 is the initial state (a set of atoms that hold true), G is the set of atoms that shall be true in the goal state and an empty plan π.
Output: A plan π or a *failure*, implying a non-existence of the plan.

```
1:  function DFSPLAN(A, s, G, π)
2:      if G ⊆ s then
3:          return π
4:      end if
5:      applicable ← {a|pre(a) ⊆ s}
6:      for all a ∈ applicable do
7:          s' ← (s ∪ eff(a)) \ del(a)
8:          π' ← π.a
9:          π' ← DFSPLAN(A, s', G, π')
10:         if π' ≠ failure then
11:             return π'
12:         end if
13:     end for
14: end function
```

The occasion of the first International Planning Competitions (Long et al., 2000b) brought shortly to popularity non-optimal approaches to search, where the *enforced hill climbing* used in the (Hoffmann, 2001) outperformed most of the competitors. The Algorithm 2 starts in the initial state and runs the *breadth-first search* (BFS) until it finds a state with a better heuristic value, in which

case it throws away all the previously explored states and starts the BFS again. Generally, the search proceeds by periods of exhaustive search, bridged by quick periods of heuristic descent. The function h denotes the state heuristic estimator that provides the expected distance of the state from a goal state. The array *pre* is a list of predecesors of states that allow to reconstruct the plan if a goal state is reached through the function $PATH$. The algorithm is incomplete, it may not find an existing plan and once again, it returns the first plan found disregarding the quality of the plan. Further, the performance of the algorithm is strongly tied to the informativeness of the provided heuristic function for evaluating the distance of states from the goal, which we shall further discuss in Section 1.2.

Algorithm 2 Enforced Hill-Climbing

Input: (A, s_0, G), where A is a set of actions, s_0 is the initial state (a set of atoms that hold true) and G is the set of atoms that shall be true in the goal state.

Output: A plan π or a *failure*, implying a non-existence of the plan.

```
1: function EHC(A, s_0, G)
2:     open ← {s_0}
3:     best ← h(s_0)
4:     while open ≠ ∅ do
5:         s ← POP(open)
6:         succesors ← {s' | ∃a ∈ A, ξ(s, a) = s'}
7:         for all s' ∈ succesors do
8:             pre(s') ← s
9:         end for
10:        while succesors ≠ ∅ do
11:            next ← POP(succesors)
12:            if G ⊆ next then
13:                return PATH(next)
14:            end if
15:            if h(next) < best then
16:                succesors ← ∅
17:                open ← ∅
18:                best ← h(next)
19:            end if
20:            open ← open ∪ {next}
21:        end while
22:    end while
23:    return failure
24: end function
```

While the plans produced by the algorithm are *satisfying* by solving the problem and they are often found comparatively faster than with other approaches, they tend to be unnecessarily long and do not hold up to expectations from a modern planning system applied for the real-world problems. The shift of interest towards the quality of plan in recent decade refreshed the approaches that provide good ratio between quality of the plan and the time spent searching for

it. One of the most widely adapted search techniques is the A* algorithm (Hart et al., 1972). We show an adaptation of the A* for planning in the Algorithm 3.

Algorithm 3 A*

Input: (A, s_0, G), where A is a set of actions, s_0 is the initial state (a set of atoms that hold true) and G is the set of atoms that shall be true in the goal state.

Output: A plan π or a *failure*, implying a non-existence of the plan.

```
1: function ASTAR(A, s_0, G)
2:     open ← {s_0}
3:     closed ← ∅
4:     while open ≠ ∅ do
5:         s ← argmin_{s'} {f(s') = g(s') + h(s'), s' ∈ open}
6:         open ← open \ {s}
7:         if G ⊆ s then
8:             return PATH(s)
9:         end if
10:        closed ← closed ∪ {s}
11:        succesors ← {s' | ∃a ∈ A, ξ(s, a) = s'}
12:        for all s' ∈ succesors do
13:            pre(s') ← s
14:        end for
15:        open ← open ∪ (succesors \ closed)
16:    end while
17:    return failure
18: end function
```

The A* algorithm is quite similar to the enforced hill climbing. It maintains an extra queue of closed states, to avoid revisiting states, and it uses a priority queue. The issue of revisiting states have not arisen in hill climbing since we always choose the state with a better heuristic value and never go back. In the A* algorithm we do not need to revisit the states as long as the heuristic h is *monotonic*. A monotonic heuristic satisfies the following conditions:

$$\forall s_g \in S, G \subseteq s_g, h(s_g) = 0, \text{ and } \forall s, s' \in S, h(s) \leq c(s, s') + h(s'),$$

where $c(s, s')$ is the cost of achieving the state s' from the state s. For a non-monotonic heuristic h we would have alter the line 15 of the algorithm and consider revisiting the closed states if we have found a cheaper path. Further, given an heuristic that does not overestimate the distance of a state from the goal (which is already satisfied by being monotonic), known as *admissible heuristic* or a *lower-bound heuristic*, the A* is a complete and optimal algorithm. The completeness comes from the fact that A* eventually explores all the states and the optimality from the organization of the priority queue. We assume we have a cost function $g(s)$ for each state s, that calculates the cost of reaching the state - e.g. the sum of costs of all actions on the path from s_0 to s, and an admissible heuristic h. Then the first goal state s we find at line 7 also determines the optimal plan π. If there was a state s' leading to a better plan π', then its cost $g(s') + h(s')$ must

have been lower at line 5 (since h does not overestimate), which contradicts the principle of picking the state with the lowest cost at each step.

A* by itself can be quite memory consuming, since it does not throw away any state. The quality of the heuristic is especially crucial, since the less informative it is, the closer A* resembles a simple breath-first search (eventually being the same if $\forall s \in S, h(s) = 0$), exhaustively exploring all paths. Various adaptation has been seen that try to improve its performance, weighted-A* is a quite common, used by one of the best performing planners in recent years (Richter and Westphal, 2008). The principle lies in increasing the significance of the heuristic estimation by multiplying it by a certain weight $w \in [1, \infty)$, the cost estimate is then calculated as:

$$f(s) = g(s) + w * h(s)$$

The increased weight directs the search towards finding the goal faster, however at the price of the optimality guarantee. Common approach is to continually decrease the weight and obtain incrementally better solution for as long as there is a time to spend.

All the three algorithmic approaches we showed in this section share in common the strong dependence on the quality of the heuristic. Given that the computation of a "perfect heuristic", that estimates the precise cost of achieving a goal, is generally as hard as the planning itself, large part of the research done today in the field of classical planning is concerned with finding new efficient heuristics. We shall investigate the state of the field in the next section.

1.2 Heuristics

In a broad picture, heuristics are a form of guiding generic strategies that help to control the problem solving. When we are concerned about solving a problem through search, we can find a variety of general heuristics that help to enhance it. A simple yet powerful example can be the *min-domain* heuristic. Imagine that we have a task to find the values for a set of variables, where for each variable we have a domain of possible values. Then *min-domain* heuristic suggests to first assign a value to the variable with the smallest domain. The motivation for the heuristic is an expectation of reduction of the size of the search tree - having less branching closer to the root generates a tree with fewer nodes and in turn implies less work for the search. While such general heuristics are widely used in planning, in this section we shall focus exclusively on heuristics specific for solving the domain-independent planning problems.

If we consider a graph where the states of the world form the nodes and the action form the arcs between the states, then the classical planning problem is finding a path from the given initial state to some goal state. The difficulty comes from the fact that the graph is exponential in the number of atoms, whose combinations explode into $O(2^n)$ states. Therefore, finding informative and efficiently computable heuristics has been motivated for long time since the beginning of the automated planning research.

The general concept of constructing a planning heuristic is the relaxation of some part of the problem, making the problem efficiently solvable, and using

the cost of the relaxed problem as an estimation cost of the original problem. To measure the quality of the heuristic we also have to take into account its computational cost. There is often a trade-off between how well the heuristic guides the search and the time for calculating it. The most common way for evaluating a planning heuristic is the comparison of the real-time performance of a planner using the heuristic with results of other planners in the International Planning Competition (Olaya et al., 2011).

Today, we can find several families of domain-independent planning heuristics. We distinguish them based on the concept they capture:

- *Delete Relaxation* is based on the idea of solving a relaxed planning problem Π^* without any delete effects in the original planning problem Π. The cost of the solution of Π^* can then be used as a heuristic h^+ for the original problem Π. However, h^+ is still NP-hard to compute (Bylander, 1994), hence an admissible estimate of h^+ formed by the heuristic h^{max} (Bonet and Geffner, 2001) is used instead. For a planning problem (V, I, G, A) the heuristic is defined as $h^{max}(s) = max_{v \in G}\, c_s(v)$, where $c_s(v)$ denotes the cost of achieving an atom v from state s. The cost is then defined recursively as $c_s(v) = min_{a \in A, v \in add(a)}(cost(a) + max_{p \in pre(a)} c_s(p)$ if $v \notin s$ and $c_s(v) = 0$ if $v \in s$. In other words, the heuristic approximates a cost of achieving an atom v as the cost of achieving the cheapest action that adds v, whose cost is approximated by the cost of its most costly precondition. The heuristic is efficient to compute, but it is not usually as informative as other approaches.

- *Reachability Relaxation* is the concept of the h^m family of heuristics (Haslum and Geffner, 2000). The heuristic approximates the cost of achieving a set of atoms by the cost achieving the most costly subset of size m. For $m = 1$ we get $h^{max} = h^1$. In Section 1.5 we shall further leverage the information provided by the heuristic h^2 for finding pairs of atoms that cannot occur together in any state of the state space. The unreachable pairs are exactly those with infinite cost estimation found by h^2.

- *Abstractions* is a general idea of mapping each state s of the problem Π to some abstract state $\alpha(s)$. Then the heuristic $h^\alpha(s)$ is the distance of $\alpha(s)$ from the closest abstract goal state. The function α is a homomorphic function, where different mappings has been proposed in the context of pattern databases (Edelkamp, 2001), merge-and-shrink abstractions (Helmert et al., 2007) and structural patterns (Katz and Domshlak, 2008).

- *Landmarks* of a planning problem are those actions and atoms that must necessarily appear in any valid plan. They have been first investigated in (Porteous and Sebastia, 2000) and several heuristics based on landmarks have been proposed in (Karpas and Domshlak, 2009). The problem of deciding whether an action or an atom is a landmark is unfortunately as hard as the planning itself (we can imagine deciding whether a goal atom is a landmark), but several polynomially computable techniques for discovering subsets of the problem landmarks has been proposed. E.g. a satisfactory condition for an atom being a landmark is that the relaxed task Π^*, stripped from effects that add the atom, becomes unsolvable. The planning approaches leveraging landmarks in a form of heuristics has seen success

in the winner of IPC 2008 (Richter and Westphal, 2008) and one of the most informative heuristic *landmark-cut* has been proposed in (Helmert and Domshlak, 2009).

We shall further focus on the h^2 heuristic. Originally, an equivalent computation appeared in the Graphplan system (Blum and Furst, 1997) and later the recursive computation has been used for an inadmissible heuristic in the HSP system (Bonet et al., 1997). The admissible heuristic h^m, as we have defined it, has been formalized in (Haslum and Geffner, 2000). The computation of h^m is polynomial in the number of atoms and actions in the system but exponential in the parameter m, hence in practical applications we do not choose m larger than 3. The heuristic values can be computed using the Generalized Bellman-Ford algorithm, shown in Algorithm 4 for m = 2.

The algorithm takes on the input the planning problem and calculates the costs of achieving all pairs of atoms, recorded in the table T. After the initialization at lines 1-10, it continually iterates through all the actions and updates the costs at lines 12-27. Being a variation of shortest-path algorithms, it runs in time complexity $O(|V| \cdot |E|)$. The resulting table of distances can further be used for approximating the distance from the goal. In Section 1.5 we shall be further interested in those pairs of atoms that have an infinite distance and use them for reformulating the planning problem.

1.3 Plan-Space Planning

So far we have defined the search in planning directly upon the state space, where states represented the nodes, actions represented the arcs between different states and the plan was a path in the graph. Such approach is called *state-space planning*. In this section we shall elaborate on a different view of the search space, where the nodes shall be *partial plans* and the arcs shall be refinement operations intended to address the *flaws* in the plan up to eventually reaching a complete plan that achieves the goal. While notions of causality among actions and their ordering were implicit in the state-space planning, we need to handle them explicitly in plan-space planning. We shall add two concepts:

- *Causal Link* captures the causality between action preconditions and action effects. Whenever we add an action a_i into a plan to support another action a_j's precondition v by a corresponding effect, we add a causal link $a_i \xrightarrow{v} a_j$. We further need to record the precedence $a_i < a_j$.

- *Ordering Constraint* of the form $a_i < a_j$ represents the precedence between two actions and further complements the causal links, where an addition of a causal link is always accompanied by an addition of the ordering constraint. Further ordering constraints may also arise through inference in the system. We say that a set of ordering constraints upon a set of actions is *consistent* iff there exists a total ordering of the actions such that all constraints are satisfied.

We can further define the partial plan as a tuple $\pi = (A, C, L)$, where A is a set of actions, C is a set of ordering constraints between actions in A and L

Algorithm 4 Generalized Bellman-Ford

Input: A triple (V, A, I), where V is a set of atoms, A is a set of actions and I is the set of initial atoms.

Output: Table of distances T.

```
1: for all p ∈ V do
2:     if p ∈ I then T({p}) ← 0
3:     else T({p}) ← ∞
4:     end if
5: end for
6: for all (p, q), p, q ∈ V do
7:     if p, q ∈ I then T({p, q}) ← 0
8:     else T({p, q}) ← ∞
9:     end if
10: end for
11: changed ← true
12: while not changed do
13:     changed ← false
14:     for all a ∈ A do
15:         c_1 ← EVAL((pre(a)))
16:         for all p ∈ add(a) do
17:             UPDATE({p}, c_1 + cost(a))
18:             for all q ∈ add(a), q ≠ p do
19:                 UPDATE({p, q}, c_1 + cost(a))
20:             end for
21:             for all r ∈ V, r ∉ del(a), r ≠ p do
22:                 c_2 ← EVAL(pre(a) ∪ {r})
23:                 UPDATE({p, r}, c_2 + cost(a))
24:             end for
25:         end for
26:     end for
27: end while
28:
29: function UPDATE(s, v)
30:     if T(s) > v then
31:         T(s) ← v
32:         changed ← true
33:     end if
34: end function
35:
36: function EVAL(s)
37:     v ← 0
38:     for i ∈ {1, ..., size(s)} do
39:         v ← max(v, T(p_i))
40:         for j ∈ {i + 1, ..., size(s)} do
41:             v ← max(v, T(p_i, p_j))
42:         end for
43:     end for
44: end function
```

is a set of causal links. While in the state-space planning the search algorithms branched only on transition between the states (representing an addition of an action into the plan), in plan-space planning, having a partial plan $\pi = (A, C, L)$ we can progress by three steps:

- Inserting an action into A.
- Adding a causal link into L.
- Resolving a *threat* to a causal link. The threat to a causal link $a_i \xrightarrow{v} a_j$ is an action a_k, $v \in del(a_k)$ that may occur after a_i and before a_j. The threat can be resolved by introducing a constraint $a_k < a_i$ or $a_j < a_k$ and we need to backtrack, if none of those is consistent.

The solution plan for a transition system $\sum = (S, A, \xi)$ and a problem $\Pi = (V, I, G, A)$ is then a plan $\pi = (A, C, L)$ such that all ordering constraints in C are consistent and every sequence of totally ordered actions in A satisfying C defines a path in the state-transition system $\sum$ from the initial state described by I to a state satisfying G.

We shall now introduce the basic schema of a search algorithm for plan-space planning and extend it later in Chapter 4. The Algorithm 5 exhaustively explores the plan-space. In each call of the PSP procedure, we first identify all the open goals (action preconditions without causal link) and threats in the current partial plan, we call them the *flaws* of the partial plan. If there are none, the plan is complete, otherwise we progress for each resolver of a flaw (insertion of an action, causal link or a constraint). The algorithm is complete, but not necessarily terminating, and the correctness has been shown in (Ghallab et al., 2004).

While the state-space was finite, but exponential in the number of atoms, the plan-space is infinite as there are no limitations on actions insertions. In that light, the PSP algorithm is not truly practical until we further introduce heuristics, symmetry breaking and some form of control of cycling. We may as well use the iterative deepening or switch to breadth-first search. It is also often hard to transfer the state-space planning heuristics for plan-space planning, because the notion of intermediate state disappears in the partial plan. On the other hand, plan-space planning provides more reasoning power over the causality decisions to the search algorithm, and is actually postponing the commitment on ordering of the actions until it is necessary. The *partial-order causal-link* (POCL) planning has attracted large amount of research effort two decades ago, resulting into systems SNLP (McAllester and Rosenblitt, 1991) and UCPOP (Penberthy and Weld, 1992). One of the key focuses has been the flaw selection strategy (Pollack et al., 1997). In the last decade we have been shown ways how to benefit from reachability and distance-based heuristics in the POCL planning (Nguyen and Kambhampati, 2001), and the competitiveness of the resulting planner VHPOP (Simmons and Younes, 2011) has shined in the third International Planning Competition (Olaya et al., 2011). The POCL approach is also better suited for supporting resource reasoning, where the ordering constraints come not only from planning the actions but also from supporting the resource constraints.

Algorithm 5 PSP Exhaustive Search

Input: A planning problem (V, I, G, A), the PSP planning procedure then takes the initial plan on the input. We create the initial plan π by introducing two artificial actions a_s and a_e, where a_s adds all atoms in I and has no preconditions and a_e has as preconditions all atoms in G and no effects.
Output: Solution plan or a $failure$.

```
1: function PSP(π)
2:     flaws ← OPENGOALS(π) ∪ THREATS(π)
3:     if flaws = ∅ then
4:         return π
5:     end if
6:     CHOOSE(f ∈ flaws)
7:     resolvers ← RESOLVE(π, f)
8:     for all r ∈ resolvers do
9:         π' ← APPLY(π, r)
10:        π' ← PSP(π')
11:        if π' ≠ failure then
12:            return π'
13:        end if
14:    end for
15:    return failure
16: end function
```

1.4 Hierarchical Task Networks

Hierarchical Task Networks (HTN) (Tate, 1977) capture the idea of planning as a process of refinement of high level plans into lower level plans. We may imagine a high level plan to be traveling from Toulouse to Prague. Going deeper, we may choose to either travel by train, car or a plane. If we choose the plane, we can refine the plan into more detail, planning that we need to buy ticket, travel to the airport in Toulouse and then back from the airport in Prague. Compared to the flat structure of the plan in classical planning, the HTN resembles forest of refinement trees, where the leaves correspond to the actions we find in classical planning. Among the planning approaches we have presented so far, HTN is the one closest to how people actually reason about actions and change.

The planning with HTN shares the same representation of states, as we have set up for classical planning in Definition 1.0.1, and the actions represent deterministic transitions between the states. Instead of searching for a way how to achieve a set of goal atoms, HTN searches for ways how to perform a set of given *tasks*. We are further given *methods* how to decompose tasks into subtasks (a subtree) and through the decompositions we recursively decompose any non-primitive tasks until all leaves are primitive tasks (actions).

We shall formally set up the terminology of HTN, using a ground representation of the concept presented in (Ghallab et al., 2004).

Definition 1.4.1. *Having a set of atoms V then a task is a triple t = (pre,eff,del), where pre,eff,del $\subseteq V$ represent the preconditions, positive effects, and negative*

effects of the task. We denote the set of all tasks as T.

For a set of atoms V and a set of tasks T we define a method $m = (t, S, C)$, where $t \in T, S \subseteq T$ and C is a set of several types of constraints:

- *Precedence constraints of the form $x < y$, where $x, y \in S$ representing that the task x must occur before task y.*
- *Persistence constraints of the form $v < x$ or $x > v$, where $v \in V, x \in S$ representing that the atom v must hold true just before (after) task x is executed.*

We denote the set of all methods as M. We say a task t is primitive iff $\forall(t_i, S_i, C_i) \in M, t_i \neq t$.

The task network is a pair w = (U,C), where U is a set of tasks and C is a set of constraints.

The tasks correspond to the actions in classical planning, with the difference that some of them can be decomposed through methods. The methods then describe the decomposition rules, where we can have multiple possible decompositions for a single task, forming the decision points. The task network keeps record of the current non-primitive tasks (leaves of the decomposition trees) and all the additional constraints that have been accumulated through decompositions. Having a task network $w = (U, C)$ and a method $m = (t_i, S_i, C_i)$ such that $t_i \in U$ then the decomposition is performed as follows:

$$\delta(w, m) = ((U \setminus \{t_i\}) \cup S_i, C \cup C_i)$$

In other words, we remove the task and add its subtasks instead, and we add all the new constraints of the decomposition. The ordering of constraints in C captures how the decomposed subtasks are related to each other (e.g. buying a ticket before getting to the airport) and the constraints on atoms represent conditions of the decomposition (e.g. if it is sunny, take a bike, if it is raining, take a car). We can now define the HTN planning problem.

Definition 1.4.2. *For a set of atoms V, an HTN planning problem is a tuple $P = (s_0, w_0, T, M)$, where $s_0 \subseteq V$ is the initial state, $w_0 = (U, C)$ is the initial task network, T is a set of tasks and M is a set of methods.*

For a problem (s_0, w_0, T, M), the task network $w_m = (U, C)$ is a solution and $\pi = (t_1, ..., t_n)$ is a plan if the following conditions hold:

- $\forall t_i \in U, \forall(t, S_x, C_x) : t \neq t_i$.
- *$U = \{t_1, ..., t_n\}$ and the ordering of the tasks in time by their indexes satisfies constraints in C.*
- *There exists a sequence of methods $(m_1, ..., m_m)$ such that*

$$\delta(\delta(...\delta(w_0, m_1)..., m_{m-1}), m_m) = w$$

- *All constraints in C are satisfied.*

In general, the HTN methods can be recursive, which gives the problem more expressiveness but also brings issues with cycling detection. The HTN planning problem is at least as hard as the classical planning problem from the Definition 1.0.1, we can imagine a set of methods that decompose each task to a pair formed by itself and any other task. The HTN problem can be solved by searching through the possible decompositions until we reach a task network consisting only of primitive tasks that satisfies all the constraints. The reader may find examples of HTN algorithms in (Ghallab et al., 2004).

HTN planning has been one of the most widely used planning approaches for practical applications (Ghallab et al., 2004). This is mainly because it allows large amount of control of the planning process by forging the decomposition methods to reflect the actual knowledge of the planning problem. Sometimes it may even resemble scripting of scenarios with alternatives. Among the widely used HTN planners we can find O-Plan (Currie and Tate, 1991) and SIPE-2 (Wilkins, 1991) that have seen many practical applications, and SHOP2 (Nau et al., 2003) also having a noticeable presence in academic community.

1.5 State Variable Representation

The efficiency of planning systems is strongly dependent on the formulation of the problem – there are many possible formulations of the same problem and planning systems behave differently on each of the formulations. At the beginning of this chapter we have defined the *classical planning problem* as a quadruple (V, I, G, A), where the world is represented by a set of atoms V, such as *"robot R_1 is at the location L_1"*. However, it is often more convenient to use a set of functions F that set the value representing the original atom, such as $position(R1) \rightarrow L_1$. As a result, we can reduce the size of the state space from $2^{|V|}$ to $\prod_{f \in F} dom(f)$, implicitly pruning the states that cannot be reached. Those functions we call the *state variables*. Formally, having a set of functions $F = \{f_1, ..., f_n\}$ the state of the world is defined as $s = (v_1, ..., v_n)$, where $v_1 \in dom(f_1), ..., v_n \in dom(f_n)$. The modification translates fluently the original planning problem (V, I, G, A) into a new problem (V, I', G', A'), where the initial state and the goal description contain instead of atoms the conditions on a value such as $position(R_1) = L_1$ and the actions A have in the same way modified the set of precondition $pre(a_i)$ and positive effects *eff(a_i)*. The negative effects in $del(a_i)$ either disappear, if the state variable is already represented in the positive effects, if it is not, the original negative effect introduces a new effect $f_i = none$ to the modified problem, representing the fact that none of the original atoms, captured by the function f_i, holds true. Obviously, the less functions we are able to translate the original atoms into, the more compact it becomes and the more efficiency we can expect to gain when planning.

To demonstrate the benefit of the compactness of the state-variable representation and the additional knowledge it can encode, let's assume an example with a ship moving between three ports. Having three atoms:

Position(Ship, Port1),
Position(Ship, Port2),
Position(Ship, Port3),

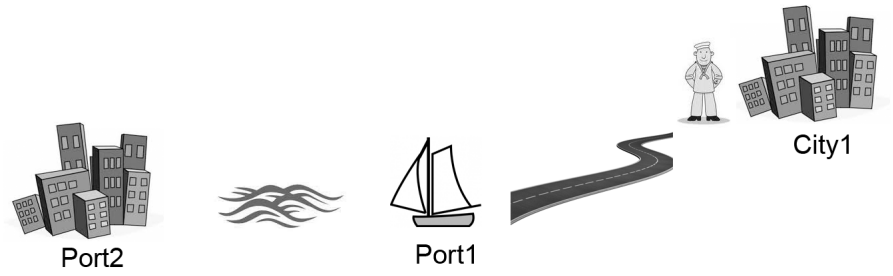


Figure 1.1: The figure illustrates a situation, where the sailor being in the City1 is mutually exclusive with the ship being at the Port2, since there is no-one else to sail it there.

where no pair of them can be true at the same time, we can encode those three facts as a single state variable:

$$\text{ShipPosition:} \rightarrow \{\text{Port1, Port2, Port3}\}$$

Although the example is very simple and the knowledge it captures could have been used by the modeller of the planning domain, other knowledge inferred from the planning problem can be non-trivial. Assume that there is a sailor, who is the only one who can sail the ship, and there is City1, where the sailor is initially located. Let Port1, where the ship is initially located, be the only accessible port from City1. Then we can deduce that if the sailor is in City1 or Port1, the ship cannot be at Port2 or Port3 (there is no one to sail the ship there). The example is illustrated in Figure 1.1. A new state variable can then combine the position of the sailor with the position of the ship as follows:

$$\text{Position:} \rightarrow \{\text{ShipPort2, ShipPort3, SailorPort1, SailorCity1}\}$$

The use of *state variables* for representing states in planning problems is an idea first formally analyzed as the SAS$^+$ representation in (Bäckström and Nebel, 1995). Although the PDDL notation (McDermott et al., 1998) was originally defined using only atom variables, it has been shown by (Dovier et al., 2007) and (Helmert, 2009) that there is a significant number of planning approaches that can directly benefit from the state variable representation by being more compact. If each state variable f_i originates from a set of pair-wise mutually exclusive (mutex) atoms (atoms that cannot occur at the same time point during the execution of any valid plan), then both the state-variable and the classical representations are equivalent. In the state-variable representation we only lose those states that could not have ever been achieved. However, to determine whether two atoms are mutex can be as hard as the planning itself. There have been several approaches for identifying mutex invariants in (Rintanen, 2000), (Gerevini and Schubert, 1998) and (Helmert, 2009), where the latest is currently being used the most, for example by the planners in the International Planning Competition (Olaya et al., 2011). Since the problem of finding the mutexes is hard, transitively, the problem of translation is hard as well.

In this section we shall thoroughly compare existing translation techniques, especially focused on the mutex generation, and propose a new technique that

often outperforms the existing ones, using the planning systems and domains from the IPC (Olaya et al., 2011) for empirical evaluation.

1.5.1 Translation

The state variables can be generally perceived as sets of pair-wise mutually exclusive atoms, we call them *mutex sets*. If we define a graph $G = (V, E)$, where V represents all the atoms and $(x, y) \in E$ if and only if x and y are mutex, then the mutex sets are the complete subgraphs (cliques) in G. Although it is possible to relax the requirement on mutex relations, e.g. allowing a few pairs to be non-mutex as seen in Seipp and Helmert (2011) to obtain larger groups of atoms, we shall consider only the cliques to be the candidates for the state variables. We are then faced with two tasks, first we need to find the cliques (which is NP-hard in general) and after that we need to select a subset of the available cliques such that all of the original facts will be covered (set covering problem, again NP-hard in general). The translation pipeline can be generally described in five steps:

1. For the given input we create the classical representation of the problem as (V, I, G, A), sets of atoms, initial atoms, goal atoms and actions. This may involve *grounding*, if the input is given in the PDDL (Fox and Long, 2003) and even breaking the existing state-variables back into atomic form, if the input is in ANML (Smith et al., 2008). The grounding is a process that produces all possible instantiations of planning operators in the problem (e.g. for action Move(x, l_1, l_2) we create all possible assignments of items into x and locations to l_1 and l_2) and does the same for the predicates in the system (creating atomic variables, e.g., empty(x) produces an atom for each possible x).

2. We discover the mutex relations through the h^2 heuristic, using the Algorithm 4.

3. Using the mutex relations we harvest the mutex cliques (mutex sets), the candidates for the state variables. Finding a maximal (the largest) clique in a general graph is a known NP-hard problem (Bomze et al., 1999; Karp, 1972), we describe our approach in Section 1.5.2.

4. We select the mutex cliques that we shall use for the state variables. Partitioning a graph G with n vertices into a minimal number of cliques is a well known Minimum Clique Cover NP-complete problem. Instead of exhaustively looking for the optimum we use an approximation algorithm presented in Boppana and Halldórsson (1992), running in $O(n/(\log n)^2)$.

5. Using the selected cliques we produce a state-variable representation as described at the beginning of this section.

1.5.2 Clique Search

Since finding a maximal (the largest) clique in a general graph is hard (Karp, 1972) we do not actually look for the largest cliques but invest the computational time into probabilistically finding a selection of large cliques, using the Algorithm 6.

Algorithm 6 Probabilistic Clique Sampling

Input: Graph $G = (V, E)$, number of cliques to be found k
Output: Set of cliques.

```
1: repeat
2:     cands ← randomPermutationOfFacts
3:     newSet ← ∅
4:     repeat
5:         candidateFact ← popFirst(cands)
6:         newSet ← newSet ∪ {candidateFact}
7:         cands ← cands \ nonmutex(candidateFact)
8:     until cands is empty
9:     record(newSet)
10: until time is up or k sets were recorded
```

The main loop of the algorithm records new sets of mutexes until an ending criterion is met. For each set, we first randomly permute the ordering of the facts (line 2), then we enter the inner loop that in each iteration chooses the first candidate (line 5), adds it to a new set of mutexes (line 6) and removes from the candidates all the facts that are not mutex with it (line 7). The algorithm can be also found in Ghallab et al. (2004) for finding minimal critical sets. The probabilistic clique-search algorithms often devote the computational time to find a clique as close to the largest as possible Richter et al. (2007). Here we do not spend any time improving but instead uniformly sample the space of cliques.

To determine a reasonable number of samples with regard to the impact on the total time of the translation, we have experimented with the number of samples related to the size of problem. Running on a selection of domains from IPC (Olaya et al., 2011) we have found out that the function $f(n) = 150 * n$, where n is the number of atoms, provides a selection of mutex sets that is rich enough for covering the atoms, while imposing less than 10% overhead for the total translation time. The dependency between the total translation time overhead and the number of samples can be seen in Figure 1.2.

1.5.3 Experimental Evaluation

The main goal of our approach is to capture more mutex-based structural information about a planning problem, and then to encode it transparently in the form of state variables, so that the existing planners using SAS$^+$ as an input could immediately leverage from this enhancement. Following on our latest contribution in (Dvořák et al., 2013), we have used a better scaling computation of the mutexes among atoms in form of the h^2 heuristic, we denote this approach as h^2*-greedy*. For the purpose of this thesis we also exclude other approaches we have considered previously, such as the Ramsey approach for covering and overcovering, whose comparison shall appear in a journal. We compare our approach to the currently dominant translation technique presented in (Helmert, 2009), which we denote as *fd-greedy*.

For evaluating the effect of the encodings on the actual computation of a

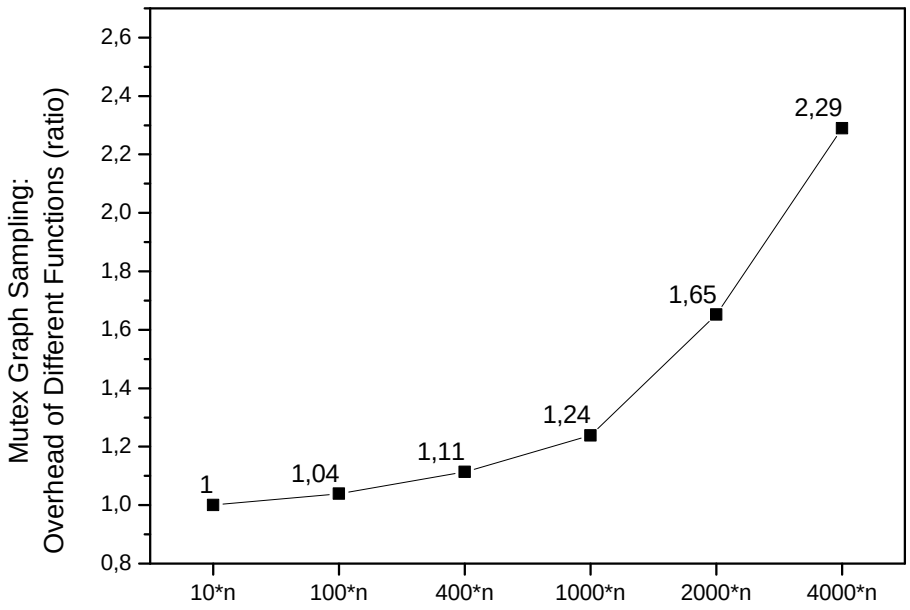


Figure 1.2: The figure illustrates the dependency between the function that defines the number of samples we take in the mutex graph and the time needed to translate all 140 testing problems from our experimental set (see Section 1.5.3 for further details) using the function. It shows the impact of mutex graph sampling on total translation time. The y-axis in the figure denotes time overhead ratio if compared to the result of using the function $10 * n$, so for example, the use of function $400 * n$ represents 11% time overhead on the total translation time compared to the use of the function $10 * n$.

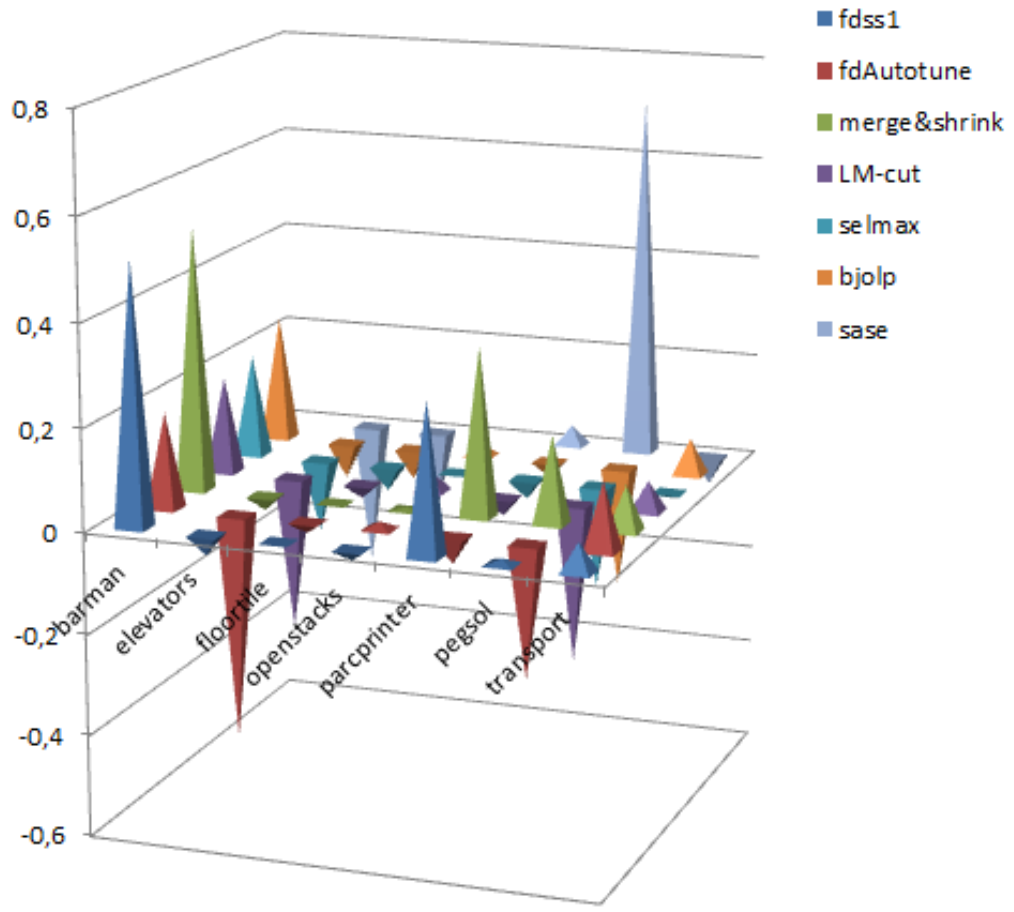


Figure 1.3: The figure shows the comparison of the total planning time between *h2-greedy* and *fd-greedy* for different planning systems on different domains. The scale on the z-axis is normalized into the interval [-1,1], where negative values represent the better performance of the *fd-greedy* and the positive ones the better performance of the *h2-greedy*.

solution we chose seven planning systems, in alphabetic order: BJOLP Domshlak et al. (2011a), Fast Downward Autotune (optimizing version) Fawcett et al. (2011), Fast Downward Stone Soup 1 (optimizing version) Helmert et al. (2011), LM-Cut Helmert and Domshlak (2011), Merge and Shrink Nissim et al. (2011), SASE Huang et al. (2010), and Selective Max Domshlak et al. (2011b). Further we use a selection of seven problem domains from the optimization track of the IPC 2011. The experiments were performed using a cluster of 15 identical Ubuntu Linux machines equipped with Intel® Core™ i7-2600 CPU @ 3.40 GHz and 8GB of memory.

For each domain and planner we have focused on the comparison of the averages of the total computational times that are formed from the time needed for translating a problem in a domain and then finding the optimal plan for it by the planner. We have used a standard 30-minutes time out per problem, having 140 planning problems in total. Further, we compare the time only for problems where a planner finds a plan for both encodings, in total, the planners have found 3 more plans using the *h^2-greedy* than when using *fd-greedy*. The relative comparison of the total times per domain are shown in Figure 1.3. We use a normalized comparison using formula:

$$f(x) = \begin{cases} x^{-1} - 1 & \text{if } x > 1 \\ 1 - x & \text{if } x \leq 1 \end{cases}$$

where the input of the function f is the division of total runtimes for all problems in a domain, *h^2-greedy/fd-greedy*.

Our initial motivation was to invest more time into the translation of the problem and observe whether it pays off in the total run-time of the planner. Looking at the barman domain, the *h^2-greedy* approach performs strictly better across all the planners, mainly thanks to better compactness of the encoding, generating on average 65% less state variables than the *fd-greedy* approach. On the other hand, the encodings in the elevators domain are practically identical and we can observe lower performance of the *h^2-greedy*, since the translation takes on average 0.365s, while the *fd-greedy* takes 0.254s. Further, planners using the SAS$^+$ directly, such as SASE and Merge&Shrink, can benefit from the compact representation much more, which we can see in the notably better performance of the Merge&Shrink and outstanding performance of the SASE in pegsol domain, where the *h^2-greedy* is able to discover very large cliques compared to *fd-greedy* (e.g., *h^2-greedy* discovers a clique of size 16, where *fd-greedy* discovers a large number of cliques of size 2). Large cliques discovered in the parcprinter domain show further benefit in the planning time of Merge&Shrink and fdAutotune.

1.5.4 Summary

We have proposed a novel method called *h^2-greedy* for constructing the finite-domain state-variable representation of planning problems. Following up on the previous work on the topic Dvořák et al. (2013) we have abandoned the computationally expensive construction of the planning graph and replaced it with a better scaling h^2 heuristic that produces an equivalent set of mutexes. Our method is fully automated, and it can provide significant speed-up on certain domains and planners, compared to today's standard method *fd-greedy*. Through

experimental evaluation we have discovered that the method is especially helpful for planning systems that take advantage of the state-variable formulation, such as SASE and Merge&Shrink and for domains such as barman, parcprinter and pegsol it constructs a more informative encoding. The translation tool is written in Java and shall be publicly available.

1.6 Lifted Representation

Both the classical planning problem in Definition 1.0.1 and the state variable representation introduced in the previous section operate with potentially large sets of atoms (state variables) and actions. Having a logistic problem of transporting a set of items I between location L by a set of cars C, the number of actions can grow as much as $|I| \cdot |L|^2 \cdot |C|$, representing the transportation of item $i \in I$ between locations $l_1, l_2 \in L$ by a car $c \in C$. The large logistic problems from IPC (Olaya et al., 2011) reach millions of actions. We call such enumerative representations to be *ground*. The ground representations are neither convenient for the initial description of the problem or practical as an input of the planner; in some cases they can be so large that they do not fit into the memory. The idea of lifting the representation is to work with schemes of atoms and actions parametrized by object constants. We shall define a lifted (unground) version of the classical problem, also known as the *classical representation* (Ghallab et al., 2004).

Definition 1.6.1. *For a set of objects C and a set of types $T = \{x \subseteq C\}$, we define (P, I, G, O) to be a classical planning problem such that:*

- *P is a set of predicates parametrized by typed objects in C.*
- *I is a set of parametrized predicates that are true at the initial state.*
- *G is a set of parametrized predicates that shall be true in the goal state.*
- *O is a set of operators, where each operator is a triple (name, param, pre, eff, del) such that name represents the unique name of the operator, param is a set of typed variables for objects, pre, eff and del are sets of predicates parametrized by the variables in param.*

The typing of the object constants helps to reduce the combinatorial explosion of predicates and actions. In general, the types can form a hierarchy based on the shared attributes, such as $Vehicle \rightarrow Car \rightarrow Hatchback$. We can imagine an example of a logistic problem, where:

- $Car = \{c_1, c_2\}$, $Item = \{i_1, i_2\}$, $Location = \{l_1, l_2, l_3\}$ and $C = Car \cup Passenger \cup Location$
- $P = \{at(o \in Item \cup Car, l \in Location)\}$.
- $I = \{at(c_1, l_1), at(c_2, l_2), at(i_1, l_1), at(i_2, l_2)\}$.
- $G = \{at(i_1, l_2), at(i_2, l_1)\}$.

- O consists of an operator $(transport, \{l_s, l_e \in Location, c \in Car, i \in Item\}, pre, eff, del)$, where:
  - $pre : \{at(c, l_s), at(i, l_s)\}$
  - $eff : \{at(c, l_e), at(i, l_e)\}$
  - $del : \{at(c, l_s), at(i, l_s)\}$

The lifted representation is the dominant representation for exchanging the planning problems today, we can find to large extent used by problems specified in PDDL (Fox and Long, 2003). However, most of the planning systems internally use the ground representation. The translation from the lifted representation to a ground representation, called *grounding*, is an interesting problem by itself. Using only a simple enumeration may lead to unnecessarily large ground representation and often other techniques, such as reachability analysis we have used in Section 1.5, help to reduce its size. The principle of lifting can be in similar way applied to the state variable representation and HTN.

2. Planning with Time

The mathematical structure of time is generally a set with a transitive and asymmetric ordering. It can be discrete, dense or continuous, bounded or unbounded, totally ordered or branching. For purpose of this thesis we rely on the structure of time as modeled by the set of natural numbers $\mathbb{N}$. In classical planning we reason about time only in terms of action ordering, qualitatively. We connect events in the world with relations such as *before*, *after* or *overlap*. These relations do not specify exactly when something will happen or how long it will take until something else happens; they are not settled in time. Quantitative notion of time puts events in the world into quantified relations such as event A happens "2-8 minutes before" event B. Most of the systems today are able to handle the quantified notion of time but the ways how reason about time differ. We shall further propose several reasoning capabilities that help distinguish the spectrum of approaches for handling time:

1. Actions with static durations (the duration of action is known before the planning starts).
2. Actions with interfering effects (e.g. two synchronized actions are required for opening the door: A) pushing the spring lock and maintaining the pressure, and B) turning the knob and pulling open the door).
3. Actions with context-based duration (e.g. the duration of a refueling action is linearly dependent on the amount of missing fuel).

We may find that while planning systems are able to technically work with the notion of time, the reasoning capabilities are often limited – e.g. not taking the total time into account, although it plays a role in the quality of plan, or not being able to find a plan if interfering effects are strictly required in a solution. For example one of the best performing planning systems SGPlan (Chen et al., 2006) abstracts time away completely from the planning phase and considers it separately once the planning has produced a plan. Other systems such as TFD (Eyerich et al., 2009) do not handle the temporal relations explicitly but adopt the state-space search in temporal environment, planning forward from the temporal beginning of the world to the end, having a complete state information at each step. Such approach is very well capable of handling 1) and 3), since the state information is complete. However interfering effects 2) are quite hard to reason with as the search commits to the past. These approaches are a form of relaxed temporal reasoning, while they show to be quite efficient in competitions such as IPC (Olaya et al., 2011) we are often faced with problems in the real-world where the temporal reasoning plays an important role. The concept used the most for handling time relation is a *Temporal Network* (Dechter et al., 1991). The temporal network is a graph (V, E), where the nodes V represent the events in time and the edges E represent temporal distances between the events. The general network allows labeling the edges with any set of possible temporal distances between two events, e.g. a set of disjunctive intervals, such network forms the *Temporal Constraint Satisfaction Problem* (TCSP) (Schwalb and Dechter, 1997). However, deciding whether there exists a solution to a TCSP (an assignment of time to events such that all temporal distances hold) is NP-complete,

an algorithm proposed in (Dechter et al., 1991) decides TCSP and runs in time complexity $O(n^3k^e)$, where n is the number of nodes, k is the maximal number of intervals on any edge and e is the number of arcs. Therefore only a few systems, such as TLP-GP (Maris and Régnier, 2008), actually use TCSP during planning. A more commonly used version of temporal network that restricts the edges to be labeled only by a single interval is known as a Simple Temporal Network (STN) (Dechter et al., 1991) and we can find it used in temporally expressive systems such as I_XT_ET (Laborie and Ghallab, 1995a) and EUROPA (Frank and Jónsson, 2003). We shall focus on STNs in the following sections.

2.1 Simple Temporal Network

We shall define the simple temporal network as follow:

Definition 2.1.1. *Simple Temporal Network is an directed graph $G = (V, E)$, where*

$$V = \{v_1, v_2, ..., v_n\}, \forall i \in \{1, .., n\} : dom(v_i) \subseteq \mathbb{N}$$

is a set of time variables and

$$E = \{e_{12}, e_{13}, ..., e_{1n}, e_{23}, e_{24}, ..., e_{n-1n}\}$$

is a set of arcs. Without the loss of generality we assume that all the arcs are oriented from a node with a lower index to a node with a higher index. We also exclude the arcs starting and ending in the same node since those are not semantically interesting. We further label the arcs with intervals $[a, b]$, where $a, b \in \mathbb{N}$. Our nodes represent time points, the arcs then represent their temporal relations such that $a_{ij} <= v_j - v_i <= b_{ij}$ for an arc e_{ij} labeled by an interval $[a_{ij}, b_{ij}]$.

The STN is a special case of a *Constraint Satisfaction Problem* (CSP) (Dechter, 2003) known as a *Simple Temporal Problem* (Dechter et al., 1991) (STP). Same as in a CSP the question we ask in the STP is if there exists an instantiation of variables in V such that all constraints, imposed by intervals labeling the arcs in E, are satisfied, formally:

Definition 2.1.2. *Instantiation of a simple temporal network (V,E) is a function $f_{STN} : V \to \mathbb{N}$ such that $\forall v_i, v_j \in V : a_{ij} <= v_j - v_i <= b_{ij}$.*

- *The network is consistent iff there exists an instantiation f_{STN}.*
- *In a consistent network (V, E), a constraint $[a_{ij}, b_{ij}] \in E$ is minimal iff $\forall x \in [a_{ij}, b_{ij}]$ there exists an instantiation f_{STN} where $v_j - v_i = x$.*
- *A consistent network is minimal iff all constraints are minimal.*

Not like in TCSPs, the problem of determining the consistency of STNs is polynomially solvable. For convenience of reasoning with time points we shall further define a partitioning of temporal variables in a temporal network.

Definition 2.1.3. *For a consistent $STN = (V, E)$ and each variable $v \in V$:*

- *B(v) is a set of variables such that* $\forall v_i \in B(v) : 0 \leq v_i - v \leq \infty$ *makes the network inconsistent. In other words, B(v) is a set of time variables that must occur strictly before* v*.*

- *BS(v) is a set of variables such that* $\forall v_i \in BS(v) : 0 \leq v - v_i \leq 0$ *is consistent with the network,* $0 + \epsilon \leq v - v_i \leq \infty$ *is consistent with the network, and* $0 + \epsilon \leq v_i - v \leq \infty$ *is inconsistent with the network. In other words, BS(t) is a set of time variables that may occur both at the same time and before* v*. Note that* $B(v) \nsubseteq BS(v)$*. The* ϵ *represents the smallest incremental step at the time scale, 1 if we represent the time as natural numbers* $\mathbb{N}$*.*

- $A(v) = \{v_i \in V, v \in B(v_i)\}$ *is a set of time variables strictly after* v*.*

- $AS(v) = \{v_i \in V, v \in BS(v_i)\}$ *is a set of time variables after or at the same time as* v*.*

- $S(v)$ *is a set of variables such that* $\forall v_i \in S(v)$ *: both* $0 + \epsilon \leq v - v_i \leq \infty$ *and* $0 + \epsilon \leq v_i - v \leq \infty$ *make the temporal network inconsistent. Those are the variables that must occur at the same time as* v*. We assume* $v \in S(v)$*.*

- $U(v) = V \setminus (BS(v) \cup AS(v) \cup B(v) \cup A(v) \cup S(v))$ *represents a set of time variables otherwise unrelated to* v*.*

Additionally, we define:

- $PB(v) = BS(v) \cup B(v) \cup U(v)$ *represents a set of time variables that may occur possibly before* v*.*

Figure 2.1 shows an example of such partitioning. The partitioning becomes useful later, when reasoning with resources in Chapter 3. In following sections we shall discuss operations we would like to perform on STNs and advantages of a selection of associated algorithms.

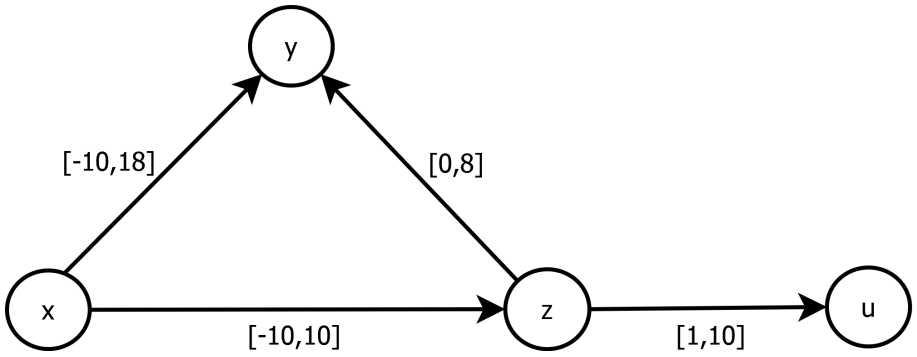


Figure 2.1: Figure illustrates and example of a simple temporal network with four time points x, y, z and u, and temporal distances between them represented as intervals $[a, b]$ labeling the arcs. Using the partitioning in Definition 2.1.3, we get that $U(x) = \{y, z, u\}, U(y) = \{x, u\}, B(y) = \{z\}, U(z) = \{x\}, A(z) = \{u\}, AS(z) = \{y\}, U(u) = \{x, y\}$ and $BS(u) = \{z\}$.

2.2 Temporal Operations

When reasoning with quantitative time the simple temporal network is one of the most intensively used structures, be it in context of planning or scheduling. Therefore, we are motivated to be interested not only in theoretical complexity but also in the practical performance of different techniques. First we identify the operations that we need to perform:

- *Consistency Check* is an operation when we ask if an STN is consistent. The property of consistency has several sufficient conditions that do not require us to actually find the instantiation. We shall investigate those later on.
- *Constraint Addition* represents an introduction of a new temporal relation between two temporal variables into the network, which leads to performing an intersection with the relation currently in the network as follows:

$$e_{ij} \cap e'_{ij} = [max\left\{a_{ij}, a'_{ij}\right\}, min\left\{b_{ij}, b'_{ij}\right\}].$$

- *Constraint Composition* represents an inference of a new constraint e_{ij} from two constraints e_{ik} and e_{kj} previously existing in the network as follows:

$$e_{ik} \circ e_{kj} = [a_{ik} + a_{kj}, b_{ik} + b_{kj}]$$

- *Retrieval of Minimal Constraint* on a consistent network is an operation of determining for two variables v_i and v_j the interval $[a, b]$ such that $\forall x \in [a, b]$ addition of a constraint $e_{ij} = [x, x]$ will not turn the network inconsistent.
- *Constraint Removal* is an operation when we completely remove the relation between two time variables. We either remove the arc or reset it into the universal interval $[-\infty, \infty]$.
- *Variable Addition* consists of adding a new temporal variable into the network.
- *Variable Removal* consists of complete removal of a variable and all associated arcs from the network.

Depending on the scenario, under which we use the temporal network, efficiency of some operations becomes more critical than for the others. In resource reasoning we often benefit from maintaining a *complete minimal network*, where all the constraints are *minimal* (for every value there is a solution) and as a result retrieving the constraints and checking consistency runs in constant time. However the update of the network with a new constraint is comparatively expensive, running in $O(n^2)$ (Dechter et al., 1991) in the worst case, where n is the number of time points. In planning scenarios, where the temporal relations are sparse, we may instead use a *sparse network*, recording only the original constraints but not the inferred constraint compositions. Adding a constraint into the sparse network runs in constant time, however checking the consistency runs in $O(n^3)$ in the worst case. For example we can compute the minimal network that guarantees consistency using the well known Algorithm 7. The algorithm enforces the

full-path-consistency of the network implying the consistency of the network as long as all the intervals are non-empty, which has been shown in (Dechter et al., 1991). For sparse networks ($|E| << |V|^2$), the consistency can be determined more efficiently using algorithms P^3C (Planken et al., 2008) and $\triangle STP$ (Xu and Choueiry, 2003) that operate on a triangulated temporal network.

Algorithm 7 Path-Consistency for Simple Temporal Network

Input: Simple Temporal Network (V, E), where $[a_{ij}, b_{ij}] = e_{ij} \in E$ represents an arc from v_i to v_j for $v_i, v_j \in V$
Output: The consistency status of the network.

```
1: for all v_i ∈ V do
2:     for all v_j ∈ V \ {v_i} do
3:         for all v_k ∈ V \ {v_i, v_j} do
4:             e_ij ← e_ij ∩ (e_ik ∘ e_kj)
5:             if a_ij > b_ij then
6:                 return inconsistent
7:             end if
8:         end for
9:     end for
10: end for
11: return consistent
```

Since in the resource reasoning in scheduling we know all the activities and the number of related time points in advance, we can allocate a single instance of the STN and any search technique such as *branch and bound* can be introducing new constraints when progressing and rolling them back when backtracking, where both operations take $O(n^2)$ (Planken, 2008) for a complete minimal network. In planning we do not know the time points in advance and we add new ones as the search progresses. The continual enlargement of the network, resulting from building up a plan, requires allocating a new space for the network. This motivates instead of having single *backtrackable network* (a network that supports rollback of the constraint insertions) to create a clone of the network at the backtrack points (the decision points during the search) and instead of rolling back the constraint insertion, we can just go back to the recorded copy of the network, where the insertion did not happen. Obviously, the sparse networks support the constraint removal directly and they are also more efficient for cloning since we are only copying the original arcs instead of all $O(n^2)$ arcs in a complete network. We shall further investigate the transition of the efficiency between sparse and complete networks with regard to their density and size in the Section 2.4. In the next section we focus on the incremental algorithms for temporal networks, which tend to be the most suited for the dynamic environment of a planning system.

2.3 Incremental Algorithms

Checking the consistency, computing the minimal network and propagating new constraints from scratch is generally more expensive than incrementally updating

the network. The nature of planning is to incrementally introduce new actions into the plan, hence the incremental techniques for STN are often more desirable. In this section we shall investigate the IFPC (Planken, 2008) for incrementally updating a complete minimal temporal network and the incremental adaption of the Bellman-Ford algorithm (Cesta and Oddi, 1996) for the sparse temporal networks. To our knowledge, those two algorithms represent the state-of-the-art for incrementally updating the simple temporal network.

2.3.1 IFPC

The Algorithm 8 takes on the input a minimal temporal network and a new constraint. Since the network is minimal, we know that the new constraint can introduce inconsistency if and only if it has an empty intersection with the original constraint in the network (line 1). Further, if the new constraint does not change the value of the original constraint, we do not need to propagate any changes (line 4). Then for every node in the network we update the triangle formed from the new constraint and the constraints connected from the chosen node to the nodes forming the new constraint (lines 9-17), this takes $O(n)$ in the number of the nodes. If any update occurs on the triangle, we record the node. Finally, we have already propagated the new constraint into every constraint connected to it, but we also need to further propagate to constraints connected to the modified constraints (lines 19-21), which takes $O(n^2)$ in the worst case. The soundness of the algorithm is shown in (Planken, 2008).

2.3.2 IBFP

Before proceeding further to the algorithm for the sparse networks we shall define the *distance graph* for the Simple Temporal Problem. For a given STP (STN) the distance graph is a directed weighted graph $G_d = (V_d, E_d)$, where V_d represents the set of temporal variables and E_d represents the set of constraints. Instead of labeling the arcs with the temporal intervals (as we did in STN), we shall add weights to the arcs. For each constraint $a_{ij} \leq v_j - v_i \leq b_{ij}$ we shall add two arcs, $e_{ij} = b_{ij}$ and $e_{ji} = -a_{ij}$. In (Dechter et al., 1991) it has been shown that the STP is consistent if and only if the associated distance graph does not have negative cycles. Given a distance graph and a new constraint the Algorithm 9 determines the consistency of the underlying STP.

The algorithm has some differences with regard to the standard implementation of Bellman-Ford with a queue. We assume there is an artificial node v_0 (the "start of the world" node) connected to every other node and we calculate two sets of shortest distances, $\{e_{0i}\}$ and $\{e_{i0}\}$ instead of one. The propagation of distances is distinguished by the marks $UB(v)$ and $LB(v)$, representing that the path to the node v has been updated and needs to be further propagated (adding v to the queue Q). Cycles in the graph imply inconsistency of the network. We detect cycles first during the propagation at lines 10 and 23, and further by constructing the *shortest path trees* in *pu* and *pl*. If there is cycle on *pu* or *pl* (line 35) then the network is inconsistent as shown in (Cesta and Oddi, 1996).

The algorithm runs in the worst-case time complexity $O(nm)$, where n is the number of nodes and m is the number of edges. However, it infers at most $O(n)$

Algorithm 8 Incremental Full Path Consistency

Input: A new constraint e'_{ij} and a minimal Simple Temporal Network (V, E), where $[a_{ij}, b_{ij}] = e_{ij} \in E$ represents an arc from v_i to v_j for $v_i, v_j \in V$
Output: The updated network or *inconsistent*.

1: **if** $e'_{ij} \cap e_{ij} = \emptyset$ **then**
2: **return** *inconsistent*
3: **end if**
4: **if** $e_{ij} \subseteq e'_{ij}$ **then**
5: **return** (V, E)
6: **end if**
7: $e_{ij} \leftarrow e'_{ij} \cap e_{ij}$
8: $P \leftarrow \emptyset, Q \leftarrow \emptyset$
9: **for all** $v_k \in V, k \neq i, k \neq j$ **do**
10: **if** $e_{kj} \cap (e_{ki} \circ e_{ij}) \neq e_{kj}$ **then**
11: $e_{kj} \leftarrow e_{kj} \cap (e_{ki} \circ e_{ij})$
12: $P \leftarrow P \cup \{k\}$
13: **end if**
14: **if** $e_{ik} \cap (e_{ij} \circ e_{jk}) \neq e_{ik}$ **then**
15: $e_{ik} \leftarrow e_{ik} \cap (e_{ij} \circ e_{jk})$
16: $Q \leftarrow Q \cup \{k\}$
17: **end if**
18: **end for**
19: **for all** $p \in P, q \in Q, p \neq q$ **do**
20: $e_{pq} \leftarrow e_{pq} \cap (e_{pi} \circ e_{iq})$
21: **end for**
22: **return** (V, E)

new constraints, representing the distances $\{e_{0i}\}$ and $\{e_{i0}\}$, while in a minimal network the number of maintained constraints is always $O(n^2)$, where n is the number of nodes.

The extension towards a removal of a constraint can be straight-forward, we can remove it from the graph, set all the distances e_{i0} and e_{0i} to a default value $[-\infty, \infty]$ and re-run the propagation algorithm on the whole graph. The authors in (Cesta and Oddi, 1996) further leverage the information contained in the dependency trees built up during the propagation and propose a more efficient constraint removal (propagating only through the dependent distances). Since in our scenarios the constraint removal is quite infrequent, we shall not include it for now.

In the following sections we shall evaluate the algorithms on our test scenarios.

2.4 Experimental Evaluation

In the Temporal Planning we usually build up a plan through an insertion of actions that are temporally related to the other actions already in the plan. The insertion of an action involves creating new time point variables in the temporal network and an addition of several constraints between the new time points and

Algorithm 9 Incremental Bellman-Ford Propagation (IBFP)

Input: A new constraint e'_{ij}, distance graph $G_d = (V_d, E_d)$
Output: Consistency of the associated STP.

```
1: Q ← {i, j}
2: LB(i) ← true; UB(i) ← true
3: LB(j) ← true; UB(j) ← true
4: while Q ≠ ∅ do
5:     u ← Pop(Q)
6:     if UB(u) then
7:         for all (u, v) ∈ EdgesOut(u) do
8:             if e_0u + e_uv < e_0v then
9:                 e_0v ← e_0v + e_uv
10:                if e_0v + e_v0 < 0 then
11:                    return inconsistent
12:                end if
13:                pu(v) ← u
14:                UB(v) ← true
15:                Q ← Q ∪ {v}
16:            end if
17:        end for
18:    end if
19:    if LB(u) then
20:        for all (u, v) ∈ EdgesIn(u) do
21:            if e_u0 + e_vu < e_v0 then
22:                e_v0 ← e_u0 + e_vu
23:                if e_v0 + e_u0 < 0 then
24:                    return inconsistent
25:                end if
26:                pl(v) ← u
27:                LB(v) ← true
28:                Q ← Q ∪ {v}
29:            end if
30:        end for
31:    end if
32:    LB(u) ← F
33:    UB(u) ← F
34: end while
35: if pu or pl contains a cycle then
36:    return inconsistent
37: else
38:    return consistent
39: end if
```

the time points in the network. Such constraints can then effect other constraints and potentially cause an inconsistency of the network, which leads to a dead-end in the search. To simulate the scenario under which the STN is used in planning we have measured the average frequency of the operations on *Check Consistency*, *Variable Addition* and *Constraint Addition* using the Filuta planning system (Dvořák, 2009) and several domains from the IPC (Olaya et al., 2011). While the number of the consistency checks can be roughly tied to the number of time points in $O(n^2)$, the number of the constraint addition and the size of constraint network (variable additions) varies significantly based on the problem. Therefore, in generating scenarios we shall fix the number of constraints checks and generate a range of constraint and variable additions. We define a *scenario* S to be a sequence of operations $(o_1, ..., o_n)$. For each scenario we generate the sequence consisting of variable additions, constraint additions and consistency checks in the following ranges of quantity:

- *Variable Addition.* We denote the number of time points as n, where $n \in [50, 250]$, incrementing by 50. Assuming that the number of time points roughly corresponds to the number of actions multiplied by two (every action being represented by the time point at the start and at the end), plans with more then 125 actions are quite uncommon, hence we do not go further.

- *Constraint Addition.* The number of constraints ranges in $[0.1n^2, 0.7n^2]$, incrementing by 0.1, the constrained time points $v_i, v_j, i < j$ are randomly picked, the values of the interval $[a, b]$ are randomly generated, where $a \in [0, 10]$ and $b \in [100 + a, 1000]$. If the insertion of the constraint causes an inconsistency of the network, we roll it back; given the design of the constraint generation, this occurs rarely.

- *Consistency Check.* We generate n^2 consistency checking operations, roughly estimating the average number of constraint checks during the planning.

For each choice of the numbers of time points and constraint additions we generate 100 sequences of operations and permute them randomly, resulting in 3500 testing scenarios. Consequently, we execute all the testing scenarios on a complete minimal temporal network using Algorithm 8 for introducing the new constraints and on a sparse temporal network using Algorithm 9 for updates. Each scenario starts from an empty network. We have implemented both algorithms and run the scenarios on a Intel i5 2.5GHz, 8GB RAM.

The results shown in the following figures are normalized comparisons between IBFP and IFCP using formula:

$$f(x) = \begin{cases} x^{-1} - 1 & \text{if } x > 1 \\ 1 - x & \text{if } x \leq 1 \end{cases}$$

where the input of the function f is the division of runtimes, IFPC/IBFC.

Figure 2.2 shows that the transition of the efficiency from the IBFP to the IFPC occurs when the number of the inserted constraints starts reaching $0.4\%n^2$ (approximately 27% density of the network). Further, the Figure 2.3 shows that the number of nodes in the network starts to benefit the IFPC starting from

the $0.3\%n^2$ constraint insertions, however for lower number of constraints the increasing number of nodes reduces the performance of the IFPC in favor of the IBFP.

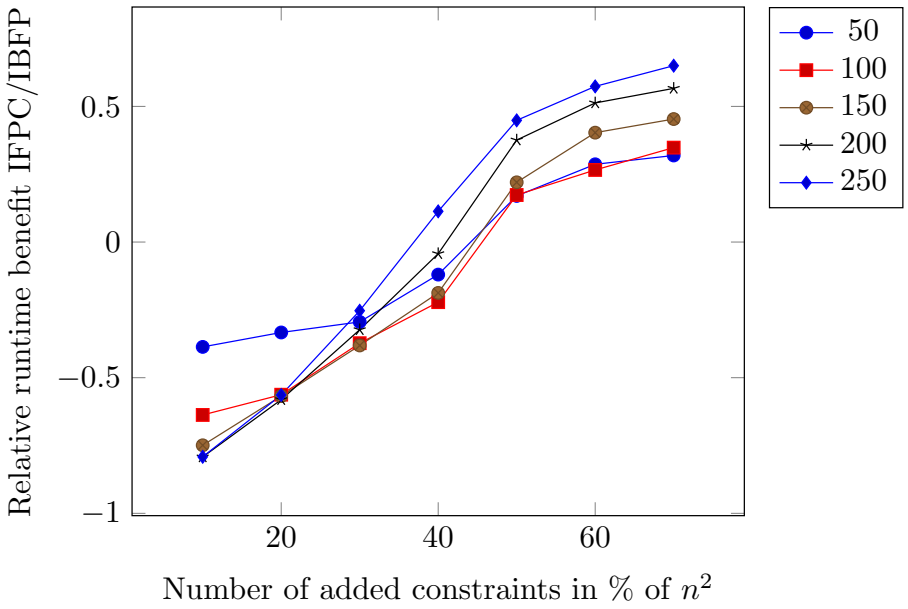


Figure 2.2: The graph shows the transition of runtime performance from the IBFP algorithm to the IFPC algorithm. The scale on the y-axis is normalized into the interval [-1,1], where negative values represent better performance of the IBFP and the positive ones better performance of the IFPC, measured as a division of the runtimes, e.g. value -0.2 shows that the IBFP is five times faster for the particular case. The colored lines show different numbers of nodes in the problems.

So far we have not taken into account the cost of copying the network and considered to have only a single instance of it. However, creating a copy of the network at the backtrack points during the search is a quite common practice, hence we further extend our set of operations by:

- *Network Duplication* creates a complete copy of the network data structure.

Using the data from the Filuta running on IPC problems, we estimate that the network duplication occurs once for every eight additions of a new constraint, therefore we extend our scenarios with the corresponding number of duplication operations and run them again. In the Figure 2.4 and Figure 2.5 we can observe that the transition point of the efficiency shifts further in favor IBFP.

The total impact of adding the network duplication operation favors the IBFP and it is more apparent with growing numbers of network nodes. Figure 2.6 illustrates the total improvement of IBFP performance, when the network duplication is added, reaching up to 12%.

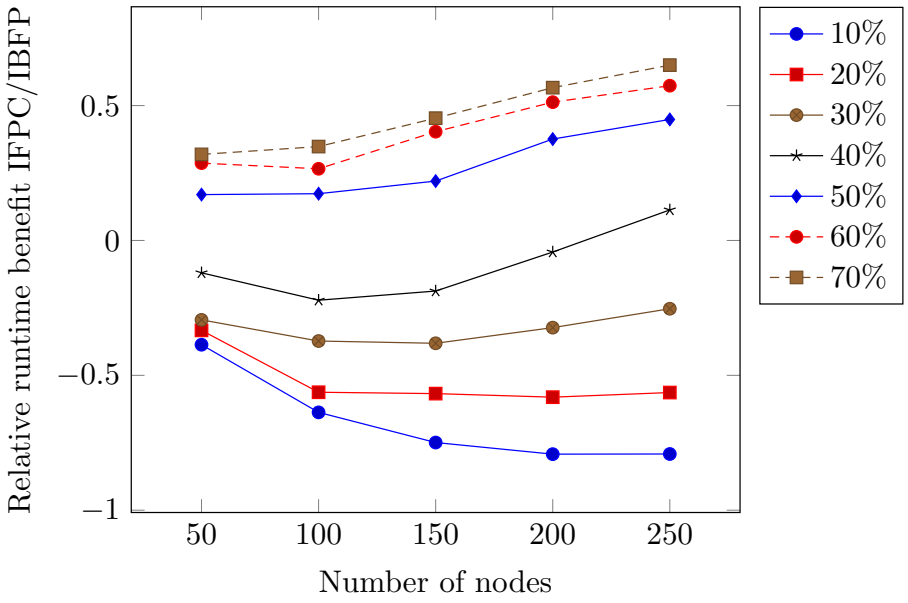


Figure 2.3: The graph shows the transition of runtime performance from the IBFP algorithm to the IFPC algorithm. The scale on the y-axis is normalized into the interval [-1,1], where negative values represent better performance of the IBFP and the positive ones better performance of the IFPC, measured as a division of the runtimes. The colored lines show different numbers of constraints in the problems.

2.5 Summary and Future Work

The Temporal Networks are a widely used concept of reasoning with quantitative time and there has been a large number of algorithms proposed in the last two decades. Due to the dynamic nature of planning we have focused in this section mainly on algorithms for incremental updates and we have evaluated their efficiency on a set of generated scenarios that reflect the expected frequency of operations in a planning system. We have identified the efficiency transition point between using a complete constraint network and a sparse network.

The natural next step of the work is the hybridization of both networks. While the sparse network is more efficient for lower numbers of time points and lower constraint densities, having the efficiency transition function, we can form a strategy that shall switch to a complete network once it becomes more efficient. Since the switching is costly (taking $O(n^3)$ to propagate all constraints into the complete network), such strategy would have to take into account the expected life-span of the data-structure.

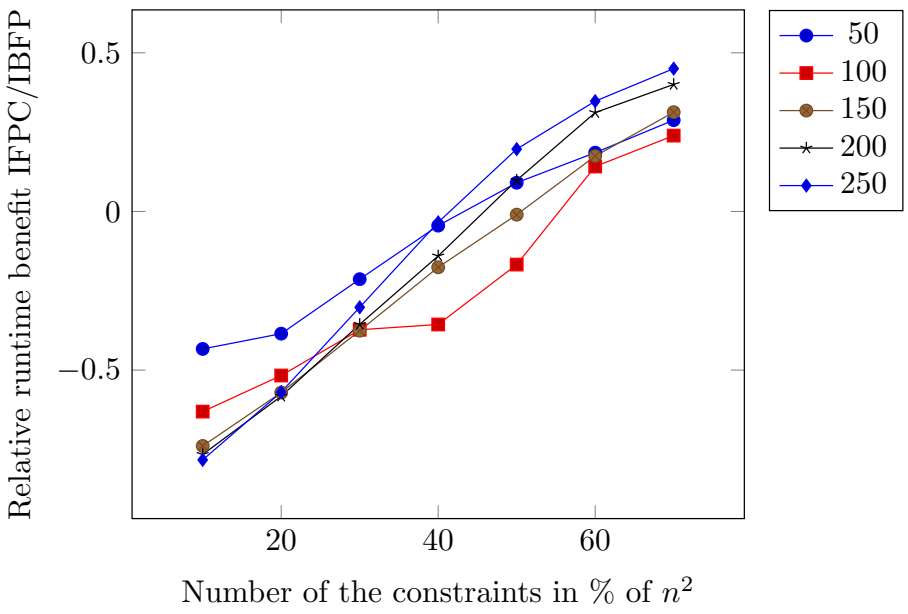


Figure 2.4: The graph shows the transition of runtime performance from the IBFP algorithm to the IFPC algorithm including the operation of duplication. The scale on the y-axis is normalized into the interval [-1,1], where negative values represent better performance of the IBFP and the positive ones better performance of the IFPC, measured as a division of the runtimes. The colored lines show different numbers of nodes in the problems.

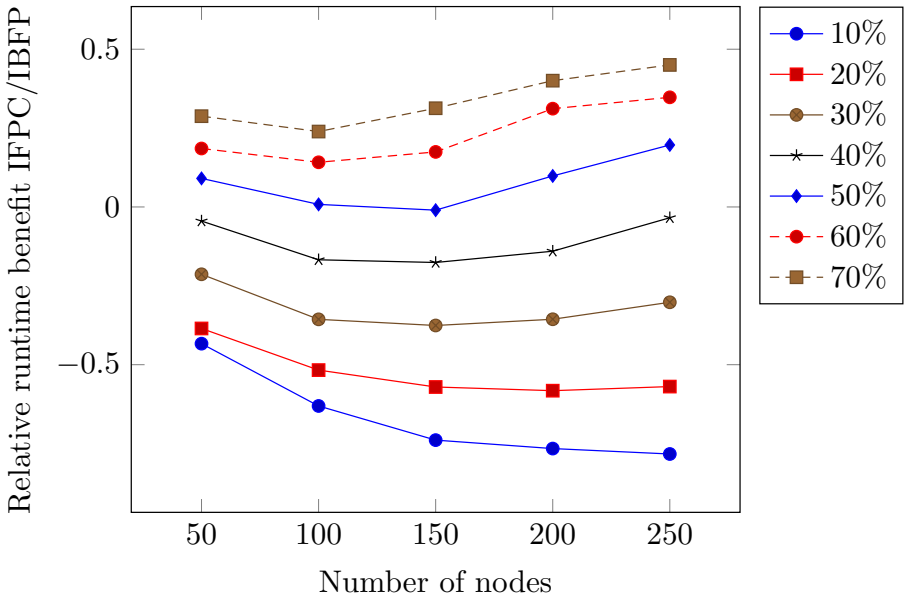


Figure 2.5: The graph shows the transition of runtime performance from the IBFP algorithm to the IFPC algorithm including the operation of duplication. The scale on the y-axis is normalized into the interval [-1,1], where negative values represent better performance of the IBFP and the positive ones better performance of the IFPC, measured as a division of the runtimes. The colored lines show different numbers of constraints in the problems.

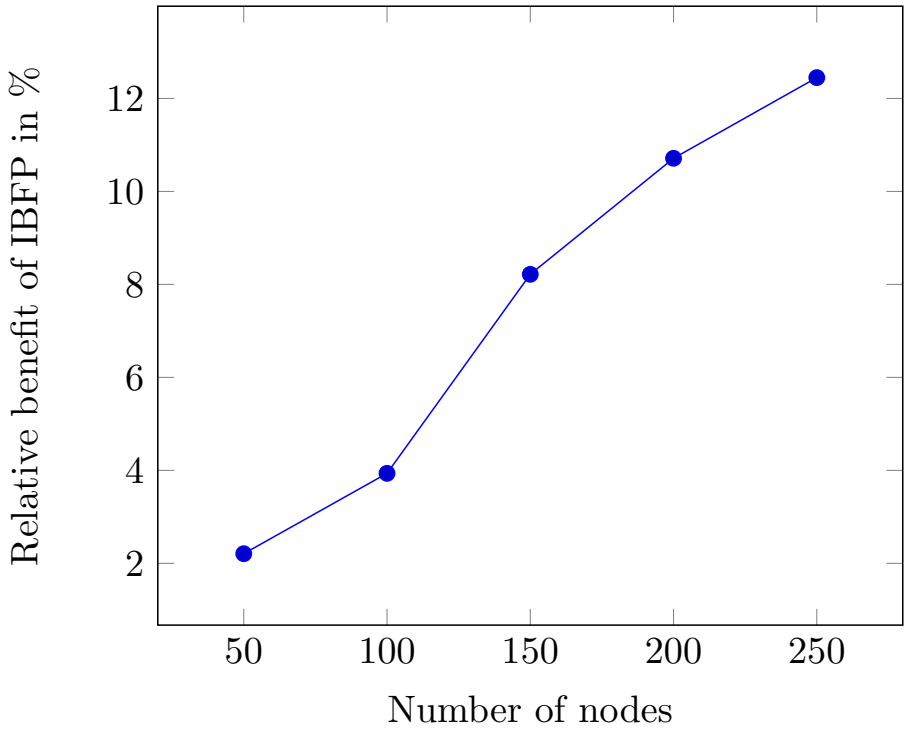


Figure 2.6: The graph shows how does the performance of IBFP over IFPC improves, when the network duplication operation is added. For each number of nodes, we denote the total runtime of IBFP without duplications as $total_{IBFP}$, and $total^d_{IBFP}$ with duplications. The totals for IFPC are denoted the same way and we show the value $\frac{total^d_{IFPC}}{total^d_{IBFP}} / \frac{total_{IFPC}}{total_{IBFP}}$ for different numbers of nodes.

3. Resources

Resource is a wide spread concept, it is "an entity that supports or enables the execution of activities" (Smith and Becker, 1997). Generally, being a resource is not a property of the object but it is a role the object plays in an activity, e.g. we can rent a car from a store as an anonymous countable entity, but then we treat the car as an individual object and become interested in its color. The role of a resource can be as simple as the representation of "being occupied", such as a single machine that can perform only a single task at a time. The resource can be borrowed in quantities, e.g. an electric outlet providing energy that cannot exceed its maximal capacity at all times, or it can be consumed and produced, e.g. the fuel in the car.

Historically, resources have been considered a domain of scheduling, in which they were extensively studied (Smith and Becker, 1997). While planning is concerned in finding a set of actions needed to achieve a goal, the scheduling problem consists of finding time and resource allocation for a set of activities. Solving many real world problems naturally requires both planning and scheduling; however separation of both processes (known as "waterfall" approach) may not always be effective, e.g. a problem with many valid plans and a few valid schedules can easily require a huge number of iterations of the planning process to reach a solution; the same issue arises when the quality metric involves resource usage (which is often expected, e.g. trying to reduce fuel consumption) and although we find a solution, the planning may uncontrollably hurt the quality of the final schedule. The problem of such sequential model is that planning itself is not enough informed how a plan should be shaped and structured to satisfy constraints later enforced in the scheduling process. As a result we can see a long standing demand for scheduling to handle planning issues and for planning to handle resources. Both directions are being explored in research and practical applications and often find a common ground in constraint satisfaction formulation (Smith et al., 2000). The main issue for the scheduling to be able to reason about planning is a different notion of activity and action. While pure scheduling assumes static set of activities, to handle planning we need to be able to consider activity occurring once, multiple times, or not at all. Further the search and heuristic techniques in planning are mostly devised for handling causality and goal achievement while resources require reasoning with constraints and optimization. When we extend planning with resources (especially with multi-capacity resources such the space in a car) we cannot easily access the current amount of resource available prior to adding an action which consumes the resource, because the amount is determined relatively to other consuming and producing actions that may not be temporally related to the new action; e.g. putting a passenger in a car consumes a unit of space, but to find out whether there has been enough space in the first place we have to determine all the possible intersections of times when other passengers are in the car and whether there has not been a moment when we have exceeded the capacity of the car. This issue does not arise in the currently popular forward-search state-space planning systems such as TFD (Eyerich et al., 2009) and LAMA (Richter and Westphal, 2008), however those planners have only a limited resource reasoning. Plan-space planning on the other hand

can accommodate resource reasoning more fluently. For example, we can insert an action, motivated by a resource conflict, into any temporal context of a partial plan and resolve a situation such as having a car refueled when it was at location A, although we have first planned a movement that consumes the fuel.

In this thesis we focus on resource reasoning techniques that help to detect *dead-ends* of the search early while following the *least-commitment* principle (not committing unless necessary). This is motivated mainly by the integration of the resource reasoning into planning, where the general idea is to channel all the decisions into a single search procedure.

In the following sections we shall categorize the resources, set up a formal model of a resource and introduce several techniques for reasoning with resources in the context of planning.

3.1 Resource categories

Resources in planning and scheduling has been distinguished in a multitude of context, we can find such ontologies in (Smith and Becker, 1997), (Laborie and Ghallab, 1995b), (Laborie, 2003b), (Smith et al., 2000) and (Long et al., 2000a). There is obvisouly no natural limitation on going further in identifying new interesting behaviour of a resource in a system and defining new classes, but we shall try to capture the current key properties we are interested in a resource behaviors, loosely following on (Dvořák, 2009).

In further text we consider a single resource to be a function $f : T \to Q$, where T represents time and the Q is some totally-ordered numeric domain, where the largest element is known as the *capacity*. The function is constructed by an introduction of new temporarily related events into the resource and the goal of the resource reasoning techniques is to determine whether the events form a *valid* function and potentially propose new events or constraints that make the function valid. We say a resource is *consistent* iff its function is valid. The validity of the function does not necessarily imply it is well defined everywhere but that it may not breach the capacity constraints whatever is the instantiation of the underlying temporal reasoning. We can imagine a simple example of the fuel in a car, after an initial refueling, we travel between several locations and consume the fuel, then refuel and travel again. Assuming we have a total capacity 60, the final function describing the development of the resource representing the fuel can be determined by a sequence of events such as $(+60, -10, -35, -10, +55, -8)$ and the function itself can be called a *schedule*.

We shall start by distinguishing the resources based on the basic events we can introduce:

- *Consume* reduces the level of the resource by a certain amount.
- *Produce* produces a certain amount of the resource increasing its level.
- *Assign* sets the current level of the resource. While *Consume* and *Produce* update the level of the resource in a relative way, *Assign* does so in an absolute way. For example, refueling a car to the maximal capacity, as in our example, or emptying a bucket.

We further distinguish resources that are:

- *Consumable*, when the resource is only consumed in the system; e.g. a limited supply of fuel in a fuel tank, that cannot be refueled. The resource capacity is usually set in the initial state and further it is only consumed in the system.
- *Producible*, when the resource is only produced in the system; e.g. some waste product of an industrial system.
- *Replenishable*, when the resource can be both consumed and produced in the system; e.g. fuel in a car, which can be refueled.
- *Reusable*, when production and consumption happen in tandem, e.g. for each consumption there exists a production. In case of reusable resources we often declare a new event *Use* that consists of a pair of temporarily constrained operations *Consume* and *Produce*.

Based on quantities that can be consumed or produced by a resource we distinguish between resources that are:

- *Discrete*, when the resource is consumed, produced, or used in discrete quantities; e.g. sitting rooms in a car.
- Continuous, when the resource is consumed, produced, or used in continuous quantities; e.g. fuel in a car.

Based on properties of capacity of a resource, we distinguish between:

- *Single-capacity*, when the resource can be thought of as one unit, which must be consumed or borrowed (used) as a whole.
- *Multi-capacity*, when the resource represents multiple units, which can be used or consumed by different operations.
- *Fixed Capacity*, when the capacity does not change over time.
- *Variable Capacity*, when the capacity of the resource is a function of time; e.g. a battery whose capacity degrades.

Additionally we distinguish between resources that are:

- *Shared*, when multiple activities can access the resource. Also known as *Batch-Capacity* resource. For example a satellite channel with given capacity that can be used by different services.
- *Exclusive*, when only a single activity can access the resource. Also known as *Unit-Capacity* resource. For example a crane that is either available or busy moving an object.
- *Single-dimensional*, when only a single attribute of the resource is considered; e.g. the number of places in an elevator.

- *Multi-dimensional*, when multiple attributes of the resource are considered; e.g. an elevator with the maximal allowed number of passengers and the maximal allowed weight.

When several resource entities appear in the system, we can further aggregate them (pool them) as an another resource of a higher level, which we further distinguish as:

- *Homogenous resource pool* is an aggregated resource composed of resources of the same type. For example a store with batteries of the same capacity.

- *Heterogenous resource pool* is a aggregated resource consisting of resources of different types and capacities. An example of such resource can be the post office, which has various trucks, cars and postmen delivering the post.

Modeling of resources is often relaxed to instantaneous events of production or consumption. However in some domains, such as refinery operations (Boddy and Johnson, 2002) we are also interested in how the level of the resource develops between those events. Based on the function that describes the development of the level of the resource in time we further talk about resources that are:

- *Piece-wise constant*, when the resource level changes instantaneously and the development between different events is constant. This is the approach adapted by most of the planning systems.

- *Linear*, when the developments of the consumption and production events in time are linear functions. There are a few systems being able to handle continuous changes of the resource level, the most recent being COLIN (Coles et al., 2012).

We shall further set up a formalism for reasoning with the resources.

3.1.1 Resource Temporal Networks

For the purpose of resource reasoning techniques interesting for planning, we shall use the *Resource Temporal Networks* introduced in (Laborie, 2003a), formalizing the concepts given in the previous section and building upon the temporal networks introduced in Chapter 2.

Definition 3.1.1. *For a temporal network N = (V,E), we define following sets of resource statements:*

- $\mathcal{R}$ *represents the set of all consumption and production events; a single event denoted by* $R(q,t)$*, where* $q < 0$ *represents a consumption,* $q > 0$ *represents a production and* t *represents a time point of a temporal network. We say that* $\mathcal{R}^+$ *denotes the set of productions and* $\mathcal{R}^-$ *denotes the set of consumptions.* $\mathcal{R}^{+-}$ *denotes a set of pairs of resource statements* $(R(q,t_s), R(-q,t_e))$*, representing lending the quantity* $q > 0$ *of the resource for the interval* $[t_s, t_e)$*. In the same way,* $\mathcal{R}^{-+}$ *represents a set of pairs of resource statements borrowing the quantity* $q < 0$*.* $\mathcal{R}_q$ *represents a set of statements that all produce, consume, lend or borrow the quantity* q*.*

- $\mathcal{A}$ *represents a set of assign events, where a single event* $A(q,t)$ *represents that the level of the resource is assigned value* q *at time* t*.*
- $\mathcal{L}$ *represents a set of conditions of the form* $L(q,t_s,t_e)$ *representing a constraint that the level of the resource must stay lower or equal to* q *over the time interval* $[t_s,t_e)$*.*
- $\mathcal{G}$ *represents a set of conditions of the form* $G(q,t_s,t_e)$ *representing a constraint that the level of the resource must stay greater or equal to* q *over the time interval* $[t_s,t_e)$*.*

We can now define the *resource temporal network* building on the temporal networks in Definition 2.1.1 and the partitioning in Definition 2.1.3.

Definition 3.1.2. *For a temporal network* $N=(V,E)$ *and the sets* $\mathcal{R},\mathcal{A},\mathcal{L},\mathcal{G}$ *using time points in* V*,* $RTN=(\mathcal{R},\mathcal{A},\mathcal{L},\mathcal{G},N)$ *is resource temporal network.*

We call the function $f_{RTN}:V\to Q$ *an instantiation of the* RTN*. We say that an instantiation* f_{RTN} *of the network* RTN *is* consistent *iff the following conditions are satisfied:*

- *The temporal network* N *is consistent.*
- $\forall(A(q_i,t_i),A(q_j,t_j))\in\mathcal{A}^2, i\neq j : t_i\in A(t_j)\cup B(t_j)$*.*
- $\forall A(q_i,t_i)\in\mathcal{A},\forall R(q_j,t_j)\in\mathcal{R} : t_i\in A(t_j)\cup B(t_j)$*.*
- *All constraints in* $\mathcal{L}$ *and* $\mathcal{G}$ *are satisfied.*

Since the temporal network N records the time points that are referred from the statements in $\mathcal{R}$, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{G}$, its consistency tells us that we can schedule the events in time. The second and the third condition removes the ambiguity of having an absolute event at the same moment as another event modifying the resource level. Finally, we require the constraints to be satisfied; since we do not focus on the most general case of the RTN, we shall define the precise satisfaction of conditions in $\mathcal{L}$ and $\mathcal{G}$ in context of specific resource reasoning techniques. Using the categorization introduced in the previous section, we can describe the most common resources appearing in planning problems using the resource temporal networks:

- $(\mathcal{R}^{-}\cup\{R(C,-\infty)\},\{G(0,-\infty,\infty)\},N)$ describes a trivial consumable resource, where we have some initial capacity C and the consumption events together may not consume more than C.
- $(\mathcal{R}_1^{+-},\{L(1,-\infty,\infty)\},N)$ is an example of single-capacity reusable resource, representing a single machine that can support only a single task, also known as the unary resource. The problem of deciding the consistency of the unary resource is already NP-complete as shown in (Finta et al., 1995). A symmetrical case is the resource $(\mathcal{R}_1^{-+}\cup\{R(1,-\infty)\},\{G(0,-\infty,\infty)\},N)$.
- $(\mathcal{R}^{+-},\{L(C,-\infty,\infty)\},N)$ is a multi-capacity reusable resource that we can find in many planning problems, representing the events of usage of limited capacity C for some time. It is often called a *discrete resource.*

Since it subsumes the unary resource and can be seen as a special case of a constraint satisfaction problem (Dechter, 2003), deciding the consistency is again NP-complete. We can formulate an equivalent symmetrical case $(\mathcal{R}^{-+} \cup \{R(C, -\infty)\}, \{G(0, -\infty, \infty)\}, N)$.

- $(\mathcal{R} \cup \{R(C, -\infty)\}, \{L(C, -\infty, \infty)\}, \{G(0, -\infty, \infty)\}, N)$ is a multi-capacity replenishable resource, one of the most general resource we encounter in planning, also known as the *reservoir*. It has some initial capacity C, initial production event that produces the capacity C and all the relative consumption and production events may not overproduce or overconsume the capacity. Deciding the consistency of a reservoir is NP-complete, it subsumes the unary resource and is subsumed by the CSP.

We can find that the planning representations introduced in previous chapters share some common concept with the resource representation. Technically, the state variables in Section 1.5 could be represented as resources $(\mathcal{A}, N)$. However, we do not gain much from such representation unless we can capture more information on how the state variables are treated and translate that into the resource terminology. One example can be finding state variables that can be treated as an unary resource (they solely change a value at the beginning of the action and change it back at the end) as was done in Filuta (Dvořák, 2009).

3.2 Resource Reasoning

Reasoning about a resource is a process of finding new knowledge through inference of available facts about the resource. E.g. having two activities being performed on a single resource that supports only one activity, we can infer that the start time of the first one must be larger than the end time of the second one or vice-versa; we call such pieces of new derived information *constraints*. The combinations of such rules and condition in which context they can be applied form the reasoning techniques. The general goal of the resource reasoning is to determine whether the constraints imposed on the resource can be satisfied, which involves searching through ways how to satisfy the constraints, interleaved with inferring new constraints. The resource reasoning is today dominantly seen as the constraint-based scheduling (Tsang, 2003).

In scheduling we can find several families of techniques that help to find new constraints in the problem, such as *timetabling* (Baptiste and Le Pape, 1995), *edge-finding* (Nuijten, 1994) and *energetic reasoning* (Lopez, 1991). They are based on establishing some necessary condition for consistency of the resource and they derive new necessary temporal constraints needed for satisfying the condition. The main issue of using them in the context of automated planning is that they consider a static set of activities (actions) and absolute positions in time. In automated planning we are operating in terms of precedence (introduced by causality) and we do not like to commit prematurely to derived constraints that may become irrelevant when new actions are introduced into the plan (e.g. a new production event introduced by an action resolves a resource overconsumption that would otherwise required a new temporal constraint). We shall describe timetabling as a representative constraint propagation technique in scheduling.

Timetabling is a propagation technique that computes for each time point t the minimal usage of the resource at that moment. We use the resource $(\mathcal{R}^{+-}, \{L(C, -\infty, \infty)\}, N)$ for describing the concept. Further, we denote $max(t)$ $(min(t))$ to be the maximal (minimal) time when t can occur. For a time point t we calculate the minimal resource level M as follows:

$$M(t) = \sum_{(R(q,t_s),R(-q,t_e))\in\mathcal{R}^{+-},t\in[max(t_s),min(t_e))} q$$

The sum counts all the quantities of usages that must necessarily occur at the time t. If there exists a date t such that $M(t) > C$ we we know the resources cannot be consistent. Further, if there exists $(R(q, t_s), R(-q, t_e)) \in \mathcal{R}^{+-}$ and time point t_0 such that:

$$min(t_e) \leq t_0 < max(t_e), \text{ and } \forall t \in [t_0, max(t_e) : M(t) + q > C$$

we derive a new constraint $max(t_e) \leq t_0$, since otherwise the resource would be overconsumed. We can apply the same principle for finding a new constraint on the variable t_s. Timetabling is one of the main techniques used today for scheduling discrete resources and reservoirs. It has good performance, naive algorithm can compute all the values $M(t)$ from scratch in $O(|V|^2)$, and for a fully instantiated temporal network it forms a sufficient condition for the consistency of the resource. However, it does not propagate anything unless the time windows $[max(t_s), min(t_e))$ start to appear in the problem and in planning we often keep temporal flexibility that prevents their appearance.

The demand on adaptation of resource reasoning techniques into planning has motivated the development of several constraint propagation techniques that operate with the precedences and not strictly with absolute temporal schedules. Those are the *energetic precedence constraints* and *balance constraints* by (Laborie, 2003b), *resource envelope* (Muscettola, 2002) and *flow balance constraints* (Frank, 2004). Further, a complete, yet often computationally expensive, approach for reasoning with resources through *minimal critical sets* has been developed in (Laborie and Ghallab, 1995b). In the following sections we go into detail with applying the minimal critical sets to discrete resources and the balance constraints to reservoirs in planning context.

3.2.1 Balance Reasoning

We shall define the *balance constraint* (Laborie, 2003b) for the reservoir resource $(\mathcal{R} \cup \{R(C, -\infty)\}, \{L(C, -\infty, \infty)\}, \{G(0, -\infty, \infty)\}, N)$, where $\mathcal{R} = \mathcal{R}^+ \cup \mathcal{R}^-$, C is the initial capacity and $N = (V, E)$ is the simple temporal network. The main idea is to compute for every time point $t \in V$ the lower and upper bound on the reservoir level just before and just after the t. For a reservoir and a time point t we shall define following values:

- $L^{<}_{min}(t)$ is the lower bound the resource level just before t.
- $L^{<}_{max}(t)$ is the upper bound the resource level just before t.
- $L^{>}_{min}(t)$ is the lower bound the resource level just after t.
- $L^{>}_{max}(t)$ is the upper bound the resource level just after t.

Using the partitioning of time points in Definition 2.1.3, we calculate the values as follows:

- $L^{<}_{max}(t) = \sum_{R(q,t_i)\in\mathcal{R}^+, t_i\in PB(t)} q + \sum_{R(q,t_i)\in\mathcal{R}^-, t_i\in B(t)} q$
- $L^{>}_{min}(t) = \sum_{R(q,t_i)\in\mathcal{R}^-, t_i\in PB(t)\cup S(t)} q + \sum_{R(q,t_i)\in\mathcal{R}^+, t_i\in B(t)\cup BS(t)\cup S(t)} q$

The values for $L^{>}_{max}(t)$ and $L^{<}_{min}(t)$ are calculated in the same way. Taking $L^{<}_{max}(t)$ as an example, it calculates the maximal resource level just before t by counting all the possible production events that may happen before t and all the consumption events that must occur before t. As a result, we can form several necessary conditions for the resource consistency, $L^{<}_{min}(t) \leq C$, $L^{>}_{min}(t) \leq C$, $L^{<}_{max}(t) \geq 0$ and $L^{>}_{max}(t) \geq 0$. Consequently, we derive new constraints based on capturing temporal relations that break the necessary conditions as follows.

Having

$$\prod^{<}_{max}(t) = \sum_{R(q,t_i)\in\mathcal{R}, t_i\in B(t)} q = L^{<}_{max}(t) - \sum_{R(q,t_i)\in\mathcal{R}^+, t_i\in U(t)\cup BS(t)} q, \text{ and } \prod^{<}_{max}(t) < 0$$

we know that there must exist some set of production events

$$P = \{R(q,t_j) \in \mathcal{R}^+, t_j \in BS(t) \cup U(t)\}$$

such that:

$$\sum_{R(q,t_j)\in P} q + \prod^{<}_{max}(t) \geq 0.$$

If such set P does not exist, we know the resource is inconsistent. If there is only a single such set P we derive a new constraint $\forall R(q,t_j) \in P : t_j < t$. If there are multiple sets, the choice of the set can be used as a decision point during the search. However, the number of generated sets can be exponential in the size of P, we can imagine a situation of picking $\binom{|\mathcal{R}^+|}{|P|}$. In context of planning, we may not even apply the new constraints (since its relevance can be invalidated by a new production event coming from an inserted action), but being aware of the non-existence of such set helps to discover a dead-end early. Similarly, an inconsistent resource may not form a dead-end, since an action that balances the resource can be introduced. The computation of the necessary condition of the balance constraints runs in time $O(n^2)$, the discovery of new constraints in $O(n^3)$, where n is the number of time points. Similarly to timetabling, once all the production events are ordered to all consumption events, the necessary condition becomes a sufficient condition for the consistency.

3.2.2 Minimal Critical Sets

We shall operate on a discrete resource $(\mathcal{R}^{+-}, \{L(C, -\infty, \infty)\}, N)$, where C is the capacity and N is a simple temporal network. The main idea is to identify all sets of possibly intersecting events that overconsume the resource. We shall first define the graph of possible intersections as $G = (V, E)$, where

Definition 3.2.1. *For a resource* $(\mathcal{R}^{+-}, \{L(C, -\infty, \infty)\}, N)$*, the graph of possible intersections as* $G = (V, E)$*, where* $V = \mathcal{R}^{+-}$ *and*

$$((R(q, t_{si}), R(-q, t_{ei})), (R(q, t_{sj}), R(-q, t_{ej})) \in E \iff \\ t_{si} \in PB(t_{ej}) \wedge t_{sj} \in PB(t_{ei})$$

The minimal critical set $S \subseteq \mathcal{R}^{+-}$ *is such a set that satisfies:*

- $\forall x, y \in S : (x, y) \in E$*,*
- $\sum_{(R(q,t_s),R(-q,t_e))\in S} q > C$*, and*
- $\forall x \in S \sum_{(R(q,t_s),R(-q,t_e))\in S\setminus\{x\}} q \leq C.$

We can see the critical sets as the cliques on the graph of possible interactions. It is intuitive that if no minimal critical sets are generated, the resource is consistent – if there was a time point t at which the resource overconsumes the capacity, the overconsumption must have been generated by a set of events $A = \{(R(q, t_s), R(q, t_e)) \in \mathcal{R}^{+-}, t \in [t_s, t_e)\}$, those events must have been pairwise possibly intersecting and as such would have implied an existence of at least one minimal critical set.

The minimal critical sets capture the potential sources of inconsistency of the resource. To eliminate a single critical set S we need to break the corresponding clique in the graph of possible intersections. This can be done by removing a single edge through addition of a new constraint that separates the events connected by the edge.

Definition 3.2.2. *For a discrete resource*

$$(\mathcal{R}^{+-} \cup \{R(C, -\infty)\}, \{L(C, -\infty, \infty)\}, N)$$

and critical set S *we define the set* $R(S)$ *of temporal constraints as:*

$$R(S) = \\ \{t_{ei} \leq t_{sj}, \exists((R(q, t_{si}), R(-q, t_{ei})), (R(q, t_{sj}), R(-q, t_{ej})) \in S^2, i \neq j, t_{ei} \in PB(t_{sj})\}$$

We call the set $R(S)$ *a set of resolvers of minimal critical set* S*.*

We distinguish several cases based on the size of $R(S)$:

- $R(S) = \emptyset$ implies that we have reached a dead-end, the resource conflict cannot be removed (not even an introduction of a new action may resolve such conflict, in contrast to balance constraint, presented in the previous section).
- $|R(S)| = 1$ is a deterministic situation, we know we can apply the resolver to the temporal network since it is the only solution for the overconsumption.
- $|R(S)| > 1$ the choice of resolver forms a decision variable for the search.

Algorithm 10 MCS-Expand

Input: Graph of possible intersections $G = (V, E)$. The algorithm assumes there is a global set MCS that contains the critical sets found by $MCSE$. The initial call of the algorithm is $MCSE((V, E), \emptyset, V, capacity)$. The value $q(v)$ denotes the consumption of the event $v = (R(q, t_s), R(-q, t_e))$.

Output: The consistency status of the network.

```
1: function MCSE((V, E), S, C, r)
2:     for all v ∈ C do
3:         if q(v) > r then
4:             MCS ← MCS ∪ {S ∪ {v}}
5:         else
6:             MCSE((V, E), S ∪ {v}, {x ∈ V, (x, v) ∈ E}, r − q(v))
7:         end if
8:     end for
9: end function
```

We can find the critical sets using the Algorithm 10 presented in (Ghallab et al., 2004).

The algorithm is a complete depth-first search that discovers all the minimal critical sets for the given graph. At each step of the recursion it chooses a new candidate $v \in C$, if it overconsumes the resource, we record the set, otherwise we filter out from the set C the nodes not connected with v and keep adding clique candidates to the set S.

Having multiple minimal critical sets generated for a resource, the sets of resolvers may have non-empty intersections with each other and some of the combinations of constraints from different resolvers may cause an inconsistency of the temporal network N. The problem of assigning a single resolver for each minimal critical set can be seen as the constraint satisfaction problem (Dechter, 2003), where the variables represent the critical sets, the domains of the variables are the corresponding sets of resolvers and a single global constraint is the evaluation of the consistency of the underlying temporal network N. While the minimal critical sets can be used to both determining the consistency and propagating new temporal constraints, the number of generated sets can grow exponentially in the size of the graph of possible intersections. Further, while determining the consistency of the simple temporal network is computable in polynomial time, deciding the existence of assignment of resolvers to the critical sets is NP-complete.

The minimal critical sets can be extended to support the reservoir resources as proposed in (Laborie, 2003b) and implemented in Filuta (Dvořák, 2009). The minimal critical sets and balance constraint introduced in the previous section can be related in their deterministic parts. If there is a set P with a single production event, the resulting temporal constraint corresponds to a resolver that is a single resolver for some minimal critical set.

4. Building a Planning System

Domain-independent planning, as a descendant of general problem solving (Newell et al., 1959), represents one of the most general abstract approaches for solving every real-world problem that can be formally described. As such, being able to solve planning problems efficiently has been a long standing goal of Artificial Intelligence. However, planning problems are generally hard to solve (Erol et al., 1995b) and we may expect them to remain hard in a foreseeable future unless new computing paradigms, such as non-deterministic quantum computing (Bernstein and Vazirani, 1993), advance to a practically applicable levels. The common ingredients of approaching a generally hard problems in context of planning are:

- *Have good performance for easy instances of the problem.* To avoid ambiguity, the hardness of a general problem ties to the theoretical time complexity of solving it, while when we speak about hardness of a particular instance of the problem, we refer to the difficulty of solving the instance in a reasonable time.

- *Fail gracefully if the instance is hard.* Hard instances can naturally emerge or can be forged on purpose and we are not able to solve them.

- *Use representation that is efficient in capturing the key aspects of the problem.* For example, the countable entities can be represented as predicates to some merit, which may be sufficient for some problems, but as soon as the presence of the operations with quantities (such as energy consumptions) increases, we need to start handling the quantities explicitly.

- *Provide a formalism that allows expressing as much helpful details as possible.* Nice example is the HTN planning that allows entering large amounts of problem specific knowledge into the planning problem. Other approaches, such as control rules and arbitrary constraints, has been used for a long time and started to appear in the academic planning language PDDL (Fox and Long, 2003).

- *Provide a lookout or approximation for the hard instances by solving a simpler problem.* For example finding an optimal plan is often more difficult than finding any plan. Anytime planning search approaches, such as used in LAMA (Richter and Westphal, 2008) and Filuta (Dvořák, 2009), start by finding any plan and then use the given residual time to improve it until an optimal plan is found or the time runs out.

The difficulty of an instance of a planning problem can spread in multiple dimensions, some of which can be identified, such as reasoning with time and reasoning with resources, and some that are not easy to identify but make the instance difficult to solve nevertheless. Our focus is to retain as much flexibility as possible across different dimensions of the problem.

In this chapter we are going to build up a new planning system FAPE with the following key properties:

- We adopt the state variable representation (Section 1.5), lift it (Definition 1.6.1) and extend with explicit time handled by STN (Definition 2.1.1. Such representation is close to the *timeline representation* used in planners such as $I_X T_E T$ (Laborie and Ghallab, 1995a) and EUROPA (Frank and Jónsson, 2003). Since we are planning with a lifted representation, we become less dependent on the size of the ground representation.

- We integrate the resource reasoning for consumable, producible, discrete resources and reservoirs and extend the representation, having resources represented as *timelines* with reasoning techniques introduced in Chapter 3.

- We further extend the representation by integrating HTN decomposition methods (Definition 1.4.1) and merge the concepts of both planning with explicit causality as in plan-space planning (Section 1.3) and decompositions.

- We pay attention on the behavior of the planning system in real-time applications by providing the following properties:

  - *Least-commitment.* We retain the temporal flexibility of the plan by not introducing temporal constraints unless they are necessary. The principle is further aided by our adoption of plan-space planning, where the actions do not need to be totally ordered, and the sparse representation of a simple temporal network, where we only check for consistency but do not introduce new inferred constraints unless necessary.
  - *Plan Repair.* Having a plan, we support a reparation of the plan if it becomes invalidated by an external cause, such as a failed action or a new temporal event. The plan reparation is an alternative to replanning that often saves computation time, which is an important aspect in real-time environment when the responsiveness of the planner plays a role.
  - *Semi-Open World Assumption.* While the planning algorithm itself operates under the closed world assumption (objects are known, no external events appear), we support an introduction of new objects, actions and temporal events into any partial or final plan. Such new objects may cause a need of repairing the plan.

In the following sections we shall first present the planning language. Then we formally define the planning problem in FAPE and follow up by presenting the search algorithm and its ingredients.

4.1 ANML

The Action Notation Modeling Language (ANML) (Smith et al., 2008) is a planning language developed in NASA as a successor of languages such as Europa modeling language (NDDL) (Frank and Jónsson, 2003) and ASPEN modeling language (Spafford and Vetter, 2012). It has a strong notion of action and state and it allows to express rich temporal constraints, state variables, numeric fluents, functions, and HTN methods. The expressiveness of ANML was the main

reason for choosing this modeling language. In turn, FAPE is the first planner that supports ANML as far as we are aware of. We support most of the major features, except for conditional effects, and we also extend the language with several syntactic constructs representing resources. There is certain amount of freedom in the interpretation of ANML and we shall now define the various elements of ANML and the interpretations we have used.

4.1.1 Types, Variables and Instances

When specifying a planning problem, we need to set up the entities that shall appear in the problem – such as robots, locations, cars and items. Such entities are often called *objects*. ANML allows to further distinguish objects into different types and provides a structural type that simplifies specification of objects, where one is contained in another, e.g. a gripper contained in a robot. It is the same principle as used in *object oriented programming*.

Typing of the objects serves the purpose of reducing the size of enumerations upon actions and functions as we have described in the lifted representation Definition 1.6.1. The types also support a single inheritance, we define them as follows.

```
type Location;
type Floor < Location;
```

Floor can be seen as special type of the Location (e.g. the location of the elevator is limited to floors, but location of the passenger can spread otherwhere), hence it inherits from it. We further support several primitive types, namely, boolean, float, integer and the enumerative type. A single state variable of some type can be declared as:

```
variable Location highestLocation;
```

Such variable then represents, for example, the highest location visited by an exploration robot. Initially, it can be the starting location of a robot and the value of `highestLocation` changes when the robot visits some location that is higher. We also define the structured types that can contain variables:

```
type Robot < object with {
  variable float energy;
  variable boolean occupied;
};
type RobotPR2 < Robot {
  variable Gripper left, right;
};
```

Variables of the structured types use the keyword `variable` and are additive with regard to inheritance. The `RobotPR2` type inherits both variables of `Robot` and any instance of the `RobotPR2` will have those variable. Now we can declare instances of types as follows:

```
instance Robot r1,r2;
instance Location bridge, elevator, appartment;
```

The `instance <type>` declaration determines the values that a variable of given type can have. An instance of a robot also forms several state variable (see Section 1.5) for each of its variables. For example, by declaring an instance of `r1`, we form a state variable `r1.energy` (equivalent to `variable float energy(r1)`), and `r1.occupied`.

4.1.2 Functions and Predicates

Functions in ANML correspond to sets of state variables (Section 1.5), where the declaration of `variable` can be seen as a function with no arguments. Their values evolve in time, unless their evolution is suppressed using a keyword `constant` – e.g. in case of static mappings such as the connectivity between locations that remains the same during planning. We can declare a function as follows.

```
function float distance(Location a, Location b);
```

The function represents the distance between two locations. Further, we consider a predicate to be a special case of the function with a binary domain. Therefore the two following declarations are equivalent:

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

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

Both declarations represent that two locations are connected. Functions can also be declared inside a structured type, making the two following declarations equivalent:

```
type LargeRobot < Robot with {
  function boolean canReach(Location l);
};

function boolean LargeRobot.canReach(Robot r, Location l);
```

A function without a parameter declared inside a structured type is equivalent to a declaration of a variable inside the type:

```
type SmallRobot < Robot with {
  function boolean hasWheels();
};

type SmallRobot < Robot with {
  variable boolean hasWheels;
};
```

Since we can encounter relations that are not meant to change during planning, we also support a constant function with the following syntax.

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

Functions and predicates represent some of the syntactic sugar available in ANML, yet we have introduced them here, since they simplify descriptions for some of our planning problem in ANML, which we shall present later.

4.1.3 Logical Statements and Temporal Annotations

In classical planning (Definition 1.0.1) actions are changing the world by adding and removing atoms, if some condition, specified as a set of atoms, was satisfied. The principle transitioned fluently into planning with state variables (Section 1.5). ANML goes further and structures the interaction with state variables into *persistent conditions*, *transitions* and *assignments*. We can declare them as follows.

```
connected(bridge, apartment) == true;
connected(bridge, elevator) := false;
r1.location == elevator :-> apartment;
```

The first represents the persistent condition that the bridge and apartment must be connected. Second one is an assignment of a value, representing that bridge and the elevator shall not be connected. The third one represents a transition from value elevator to value apartment for the state variable `r1.location`.

To capture explicitly the absolute and relative temporal allocation of different statements, we further annotate them with temporal intervals that specify, when the statement occurs.

```
[all] s;
[start,end] s;
[start-5,end+1] s;
[5,80] s;
[start,end]{
  s1;
  s2;
  s3;
};
```

In the declaration above, `s`, `s1`, `s2`, and `s3` represent any logical statements. The temporal annotation is defined as an interval `[a,b]`, where `a` $\leq$ `b`. Special variables `start` and `end` relate the evaluation of the interval to the context in which the statement appears. If it appears inside an action, the `start` and `end` represent the start time of the action and the end time of the action. Outside of an action, `start` (`end`) relates to the global beginning (end) of the world. Temporal annotation can use simple arithmetics with integer constants, + and −. The temporal annotations can be either relative to the `start` and `end` (such as `[start+1,end-1]`, or absolute (such as `[5,8]`). Further, the temporal annotation can be shared across several logical statements, encapsulating them in parentheses.

We further support a flexible temporal annotation, where `s` is a statement.

```
[start,end] contains s;
```

The annotation represents that the statement s must occur between `start` and `end`.

4.1.4 Resources and Numeric Fluents

Resources in planning can be considered to be state variables, whose domains are totally ordered and support statements that change their value in relative way. As such, the condition on the resources are also richer, supporting statements take into account the ordering. Following on the Chapter 3, we support several resources. `replenishable` corresponds to a reservoir resource, `consumable` is a simple resource that only gets decreased during planning, `reusable` corresponds to a discrete resource and `producible` is a simple resource that only gets generated during planning. We declare them as follows.

```
replenishable float [0.9, infinity] energy(Robot r, Battery b);
consumable integer fuelTank;
reusable float capacity(Robot r);
producible integer [0, 8] waste;
```

The resources can be declared both inside and outside of a type with the same semantics as the functions in the Section 4.1.2. In fact, the keywords for resource classes substitute the keyword `function`, adding more detail on how the function behaves in the system. The domains of the resource can be declared as both continuous or discrete, using either integer or float, and they can be further restricted to an interval. Note that the keywords for resources classes are not a part of the original ANML specification (Smith et al., 2008).

We further support the following resource statements, where `x.energy` can be used as a shortcut for `energy(x)`.

```
waste() >= 50;
energy(r2) <= 50;
r1.energy := 1;
r1.energy :use 10;
r1.energy :consume 10;
r1.energy :produce 10;
```

Some resources support only some of the statements corresponding to their model as described in Chapter 3. For example, the reusable resource (discrete resource) supports only the statement `use`. Its initial level corresponds to its maximal capacity in the declaration. The resource statements can be temporarily annotated in the same way as shown in Section 4.1.3.

The general numeric fluents `float` and `integer` support the same statements as resources. However, there is no reasoning attached to them and the statements are solely evaluated. It is always better to declare a resource instead of a numeric fluent.

4.1.5 Actions

The actions in ANML are collections of statements (both logical and resource statements) parameterized by the typed objects with possible decompositions consisting of other actions. We declare the actions as follows.

```
action Travel(Person p, City a, City b) {
```

```
  [start] p.location == a;
  [all] p.busy :use 1;
  :decomposition{
    ordered(
      BuyTrainTicket(p,a,b),
      UseTrain(p,a,b);
    );
  };
  :decomposition{
    UseCar(p,anyCar,a,b);
  };
};

action BuyTrainTicket(Person p, City a, City b) {
  start = end - 10;
  [end] p.money :consume 200;
}

action UseTrain(Person p, City a, City b) {
  start = end - distance(a,b);
  [all] p.location == a :-> b;
}
```

In HTN planning (Section 1.4) we have defined actions and methods, how to decompose the actions into other actions. ANML allows to specify different methods how to decompose a single action as a part of the action definition, using keyword `decomposition`. Action `Travel` contains two decompositions, where only one gets applied. We can either travel by a train, in which case we need to buy a ticket, or we use a car. The actions of the decomposition can be temporarily related to each other by ordering them through combinations of predicates `ordered` and `unordered`. The duration of the action can be specified as a generic temporal constraint referring to its `start` and `end`. The parameters of the decomposed action are passed to the actions contained in the chosen decomposition (we bind them). There can be parameters that are not bound, such as `anyCar`. The statements in an action and its decomposition cannot be redundant – e.g., setting the location of the person both in `Travel` and `UseTrain`. We use an implicit temporal constraint that for a given action, executing over interval $[x, y]$, all the actions in its decomposition must execute in $[x, y]$ as well.

Generally, the decompositions of actions can be recursive, e.g. an action can reference itself in its decomposition.

4.1.6 Initial Task Network and Goals

By merging plan-space planning and HTN planning, we adopt both approaches for specifying what should be achieved by the plan. The concept of specifying goals both as action decomposition and achievement of some condition can be seen as generalization of HTN. Recently, it also appeared in the planning system Godel (Shivashankar et al., 2013).

We can specify goals as persistent statements or transitions appearing not necessarily at the end of global time as follows:

```
goal [end]{
  r1.location == apartment;
}

[end-10,end] r2.location == apartment :-> bridge;
```

While the `goal` keyword is present in ANML, it currently does not imply any specific treatment. In the eyes of the planning system, both statements above, specified in the global context, are considered to be goals.

To support the specification of the initial task network in ANML, we declare a `Seed` action as follows.

```
action Seed(){
  :decomposition{
    unordered(
      Travel(Thomas,Prague,Toulouse),
      Travel(Jan,anywhere,Mumbai)
    );
  };
};
```

The `Seed` action has no parameters and contains a single decomposition. The decomposition consists of the initial actions in the task network that are partially instantiated by objects. Note that the `Seed` action is a construct introduced by FAPE, and it is considered to be a keyword. While the original proposal of ANML (Smith et al., 2008) does not specify how the initial task network is entered, an unpublished ANML manual contains a concept of *goal actions* that may serve the same purpose. Therefore, we expect in near future to change the syntax in sake of keeping FAPE up-to-date with the ANML.

4.1.7 Expected Events

Sometimes we need to represent a temporal evolution of some arbitrary entity which is neither under the control of the planner nor it is meant to be a goal. We can imagine a problem of representing the daylight. For this purpose we define a special action `DeusExMachina` that can be applied only once and it is introduced into a plan through the action `Seed`. The behavior of the action is constructed in ANML as follows.

```
variable boolean deusApplied := false;
variable boolean dayLight = false;
action DeusExMachina(){
    [all] deusApplied == false :-> true;
    [7,7] daylight == false :-> true;
    [20,20] daylight == true :-> false;
  };
};
```

The variable `deusApplied` guarantees that the action `DeusExMachine` gets to be applied only once. Further, we set up a sunrise at 7am and a dawn at 8pm. This construction is a natural part of the ANML language and more sophisticated applications may be developed. Note that `DeusExMachina` is not a keyword, such as `Seed`, since the action is treated as any other action and the behavior is determined by ANML. However, to our knowledge, there is not yet a way to express infinitely cycling events, such as "every 10 minutes the door open for 2 minutes". Therefore, the arbitrary events must be specified explicitly and enumeratively.

Note that in the case of daylight, we can also declare the behavior using simply the statements

```
[7,7] daylight == false :-> true;
[20,20] daylight == true :-> false;
```

in the initial problem. Yet, the `DeusExMachina` construct provides an encapsulation that may become useful for expected events, that may change. E.g. having train schedules as a part of the initial problem, we need to replan from scratch if the schedules change. But if we have an action that encapsulates the schedules in the same way as `DeusExMachina`, we may fail such action, removing it and all its statements from the plan, provide a new action with alternative schedules, and try to repair a plan.

4.2 Representation

In this section we shall build up the representation of the planning problem in FAPE using most of the concepts introduced in the previous sections.

4.2.1 Lifted State Variables

We shall use the concept of state variables describing the state of the world as an assignment to those variables and lift the representation using the object variables. We shall be planning with *parameterized state variables*, which are a form of variables for state variables. The parameterized state variable represents a state variable, we may not know which one, but we know its type. The parameterized state variables use object variables as their parameters and by reducing the domains of the object variables the set of possible state variables it represents reduces. For example, we can have two state variables $r1.location$ and $r2.location$, then a parameterized state variable $location(x)$, where $x \in \{r1, r2\}$ is an object variable, may represent either $r1.location$, or $r2.location$. Then if the domain of x gets reduced to $\{r1\}$, we know that $location(x)$ represents $r1.location$. The concept allows us to maintain a large amount of flexibility when planning by postponing the decision on usage of specific objects. For example, we can plan a transportation task, without deciding which car shall be used for the transport.

Following on the lifted representation in Definition 1.6.1, we shall define the objects, object variables, constraints upon them and their instantiation. To simplify the notation, we shall represent the types in ANML as unary constraints on the domains of the object variables.

Definition 4.2.1. *We say O is a set of objects that consists of:*

- *a finite set of object constants,*
- *real numbers $\mathbb{R}$,*
- *boolean constants true and false.*

We define K to be a set of object variables, ranging over O, denoted as $\forall x \in K : dom(x) \subseteq O$.

For a set of objects O and object variables K, we say that $\zeta : K \to O$ is an instantiation of the object variables.

We further define a unary constraint $dom(x) \subseteq Y$, where $Y \subseteq O$, and binary constraints $x = y$ and $x \neq y$, where $x, y \in K$.

For a set of objects O, object variables K and constraints C, we say an instantiation ζ is consistent with C iff the following conditions are satisfied:

- $\forall x \in K, \zeta(x) \in \bigcap_{(dom(x) \subseteq Y) \in C} Y$.
- $\forall x, y \in K, (x = y) \in C : \zeta(x) = \zeta(y)$.
- $\forall x, y \in K, (x \neq y) \in C : \zeta(x) \neq \zeta(y)$.

The definition is the same as we can find for *contraint satisfaction problems*, with three types of constraints that shall play a role in the planning system. The unary constraint is used mainly for constraining the domains of variables with correspondence to the types that are specified in ANML. The objects themselves correspond to the ANML instances. We further represent the real numbers $\mathbb{R}$ used for the type Float and Integer, and $true$ and $false$ for the type Boolean.

The parameterized state variables use the object variables as parameters. Further, their values are object variables as well. We define them as follows.

Definition 4.2.2. *For a set of objects variables K, n-ary parameterized state variable is a function $f : K^n \to K$.*

We say a parameterized state variable $f(x_1, .., x_n)$ is fully instantiated (ground) iff $\forall i \in \{1, .., n\} : |dom(x_i)| = 1$. We denote it as f^i.

We say that a ground state variable $f^i(y_1, .., y_n)$ is an instance of a parameterized state variable $f(x_1, .., x_n)$ iff $\forall i \in \{1, .., n\} : y_i \in dom(x_i)$.

We say two parameterized state variables f_k and f_l unify iff there exists a ground state variable f^i such that f^i is an instance of both f_k and f_l.

For a set of parameterized state variables $F = \{f_1, ..., f_n\}$ we define a set $F^i = \{f_1^i, .., f_m^i\}$ as a set of all ground instances of state variables in F. We further define $s = (y_1, .., y_m)$ to be the state of the world, as an assignment of values to the state variables F^i. The state space is formed by all the possible assignments of values to functions in F^i.

We can notice that the set of state variables F^i corresponds to the set of state variables as defined in Section 1.5. Further, we have lifted the representation by parametrizing the state variables, in a similar fashion as seen in Definition 1.6.1.

The parameterized state variables correspond to the abstract descriptions of state variables formed by declarations of variables in ANML. The declarations of instances in ANML then determine the objects used for instantiating the parameterized state variables and the state of the world is an assignment of values to all of the instances in F^i. In practice, we do not need to enumerate the set of state variable F^i (which can be large), but we shall be planning with the parametrized state variables, where each of parametrized state variables eventually becomes ground, representing some state variable in F^i. On the other hand, some state variables in F^i may not even become represented by any parametrized state variable, because they were not relevant to the developed plan.

We can use the example of the type Robot used in the previous section. Having a `variable boolean canReach(Location l)` declared in the `type Robot` and a declarations `instance Robot r1,r2` and `instance Location l1,l2`, we form a parameterized state variable `canReach(Robot r, Location l)` and the corresponding state variable instances would be

`canReach(r1,l1)`,`canReach(r1,l2)`,`canReach(r2,l1)` and `canReach(r1,l2)`

We assume the reachability can change, if it doesn't, the declaration of the variable could be constant. Note that in practice, we do not need to enumerate the set F^i completely, but only represent those state variables, whose value is relevant for planning (there exists a statement that manipulates or requires the value).

We shall also distinguish *logical* and *numeric* state variables. The numeric state variables are those whose domains are defined as totally ordered sets with arithmetic operations + and −, in ANML declared as `float` and `integer` variables and functions, and correspond to the resources as described in Chapter 3.

4.2.2 Temporal Statements

In FAPE we use a simple temporal network (Definition 2.1.1) to maintain the temporal relations between moments in time. We shall describe the changes and requirements on state variables by a set of temporally annotated statements, such as "the fuel in a car1 reduced by 20 units at the time point t", and use the time points maintained by the temporal network in these statements. Such view allows to represent a great amount of temporal flexibility, although we use only a single interval to relate two moments (time points) in time as discussed in Chapter 2.

In the following definition we use *time points* and *temporal constraints* as defined in the context of temporal networks.

Definition 4.2.3. *For sets of objects O, object variables K, temporal network $N = (V, E)$ and a parameterized state variable $f(x_1, ..., x_n)$, we define a temporal statements as follows:*

1. *Assignment. $A(f, t, x), x \in K$ represents an assignment of the value determined by the object variable x, occurring at the time determined by the time point t into a parameterized state variable f. We denote the set of assignment statements as $\mathcal{A}$.*

2. *Persistent Condition.* $P(f, t_s, t_e, x), x \in K$ *represents an requirement that on the interval* $[t_s, t_e]$ *the value of the parameterized state variable* f *must remain at the value determined by the object variable* x*. We denote the set of persistent conditions as* $\mathcal{P}$*.*

3. *Transition.* $T(f, t_s, t_e, x, y), x, y \in K$ *represents a change of the value over the interval* $[t_s, t_e]$ *from the value determined by* x *to a value determined by* y*. The value is undefined on the interval* (t_s, t_e)*. We denote the set of transition statements as* $\mathcal{T}$*. We can consider the assignment* $A(f, t, x)$ *to be a special case of transition* $T(f, t, t, y, x)$*, where* $dom(y) = O$*.*

4. *Relative change.* $R(f, t, x), x \in K$ *represents a relative change of the value of* f*, reducing or increasing it by value determined by* x *at the time* t*. A set of relative changes is denoted as* $\mathcal{R}$*.*

5. *Condition.* $L(f, t_s, t_e, x)$ *and* $G(f, t_s, t_e, x)$*,* $x \in K$ *represent conditions on the value of* f *over the interval* $[t_s, t_e]$*, where* $L(f, t_s, t_e, x)$*, respectively* $G(f, t_s, t_e, x)$*, represents that the value of* f *is lower, respectively higher, or equal to the value determined by* x*. A set of lower conditions is denoted as* $\mathcal{L}$ *and a set of greater conditions as* $\mathcal{G}$*.*

   For a statement s *of any type, we say that* $var(s) = f$*, is its parameterized state variable.*

The statements 1) 2) and 3) are related to *logical state variables*, while the statements 1), 4) and 5) relate to *resource state variables*. It is important to notice, that the values of statements are object variables, not the actual values. Such concept allows us to be less committing, when we reason about causality between statements on logical state variables. For example, having two transition

$$T(r1.location, t_s, t_e, x, y) \text{ and } T(r1.location, t'_s, t'_e, x', y').$$

We say that the first transition can *support* the second transition, if $t_e \leq t'_s$ and $dom(y) \cap dom(x') \neq \emptyset$.

Further, according to Definition 3.1.1, which introduced the absolute statements, relative statements and conditions upon a single resource, we have formulated statements 1), 4) and 5) by extending them with a parameterized variable that determines the specific resource. Additionally, we use a strong assumption that the domains of values of all statements upon resource variables are singular, which is the same assumption as used in (Laborie, 2003a). In other words, if f represents a resource state variable, then

- $\forall A(f, t, x) \in \mathcal{A} : |dom(x)| = 1$,
- $\forall R(f, t, x) \in \mathcal{R} : |dom(x)| = 1$,
- $\forall L(f, t_s, t_e, x) \in \mathcal{L} : |dom(x)| = 1$, and
- $\forall G(f, t_s, t_e, x) \in \mathcal{G} : |dom(x)| = 1$.

In the following section we shall describe how we relate different temporal statements to each other using *timelines*.

4.2.3 Timelines

We can already describe that a value of some parameterized state variable changed, or had some set of values (described by a domain of an object variable), at some moment in time. To describe how a single state variable evolves in time, we shall we shall be aggregating the temporal statements into timelines. A single timeline consists of temporal statements, whose parameterized state variables unify, and the statements satisfy some additional conditions. Each timeline represents the development of a single state variable, which is determined as an intersection between parametrized state variables of the statements in the timeline. Eventually, every timeline shall represent exactly one state variable. The planning process will be heavily based on merging different timelines together. We can imagine an example of such merging with two timelines

`{x.location == subway}` and `{y.location == apartment :-> subway}`.

The merging of the two would involve adding temporal precedence constraint (the value `subway` must be set up, before it is required), binding constraint `x = y` and creating a single timeline as an union of both. We shall now define the concept of timelines formally.

Definition 4.2.4. *For sets of objects O, object variables K, object constraints C, temporal network $N = (V, E)$ and a set of state variables F^i, we define a timeline $\phi = \{s_1, .., s_n\}$, where:*

- *$s_1, .., s_n$ are temporal statements,*
- *all the time points contained in $s_1, .., s_n$ are the time points of temporal network N,*
- *the parameterized state variables $f_1, .., f_n$, contained in $s_1, .., s_n$, have all in common at least one instance $f^i \in F^i$. In other words, we ensure that there exists at least one state variable, whose temporal development is represented by the timelime ϕ.*

We say a timeline is logical if all the parametrized state variables in its statements are logical state variables, we say it is a resource timeline, if they are resource state variables.

We say that a resource timeline $\phi = \{s_1, .., s_n\}$ is consistent iff following clauses hold:

- *There exists an instantiation of object variables ζ such that all object constraints in C are satisfied and all the parameterized state variables $f_1, .., f_n$ in statements represent the same instance $f^i \in F^i$. We say such ζ is a consistent object instantiation with regard to ϕ.*
- *There exists a consistent instantiation ζ such that there exists a consistent instantiation f of the resource temporal network $R = (\mathcal{R}, \mathcal{A}, \mathcal{L}, \mathcal{G}, N)$ in terms of Definition 3.1.2, where $\phi = \mathcal{R} \cup \mathcal{A} \cup \mathcal{L} \cup \mathcal{G}$.*

We say that a logical timeline $\phi = \mathcal{T} \cup \mathcal{P}$ is consistent iff:

1. *There exists a consistent instantiation* ζ *with regard to* C *and there exists an instantiation* f_{STN} *of the temporal network* N.

2. $\forall P(f, t_{si}, t_{ei}, x), P(f, t_{sj}, t_{ej}, y) \in \mathcal{P}, i \neq j$: *one of the following holds:*

   - $t_{ei} \leq t_{sj}$ *is consistent with* N,
   - $t_{ej} \leq t_{si}$ *is consistent with* N, *or*
   - $x = y$ *is consistent with* C.

   In other words, two persistent conditions cannot intersect unless they require the same value.

3. *The transition statements in* $\mathcal{T}$ *are totally ordered with regard to* N *and having a sequence*

   $$(s_1, .., s_k), s_i \in \mathcal{T}$$

   that reflects the total ordering, following holds:

   $$\forall i \in \{2, .., k\} : s_{i-1} = T(f, t_s, t_e, x_{i-1}, y_{i-1}), s_i = T(f', t'_s, t'_e, x_i, y_i) : \\ dom(y_{i-1}) \cap dom(x_i) \neq \emptyset$$

   In other words, totally ordered transitions form a sequence where each transition supports the next transition in the sequence.

4. $\forall P(f, t_{si}, t_{ei}, x) \in \mathcal{P}, x_j = T(f, t_{sj}, t_{ej}, x', y') \in \mathcal{T}$: *following conditions are satisfied:*

   - *Either* $t_{ei} \leq t_{sj}$ *or* $t_{ej} \leq t_{si}$ *holds with regard to* N.
   - *If* x_j *is the latest transition such that* $t_{ej} \leq t_{si}$ *is consistent with* N, *then* $x = y'$ *is consistent with* C.
   - *If* x_j *is the earliest transition such that* $t_{ei} \leq t_{sj}$ *is consistent with* N, *then* $x = x'$ *is consistent with* C.

   In other words, a persistent condition and a transition cannot intersect and the value of the persistent condition must be compatible with the two transitions that surround it.

We say a consistent logical timeline is supported if the earliest transition statement $x_1 \in \mathcal{T}$ *is an assignment statement,* $x_1 \in \mathcal{A}$. *Note that by design of timeline merging operations introduced later, there can never be a timeline, where the earliest transition is an assignment statement preceded by a persistent condition.*

We say a timeline ϕ *is consistent if it is either a consistent resource timeline or a consistent logical timeline.*

We say a timeline ϕ *unifies with timeline* ϕ' *iff there exists ground state variable* f^i, *which is an instance of every parametrized state variable in statements in* $\phi \cup \phi'$.

We can see a timeline as a set of parameterized statements that describe an evolution of some state variable in time. A timeline may not be tied to a specific state variable, e.g. we can plan a transportation of a certain item between two certain location, by a car that remains unknown until we bind the corresponding object variable with a particular instance of the Car type. In the same way, we know that the fuel in some car has been consumed, but binding that event to a specific resource may occur later. The combination of the set of statements and constraints has also been known as a *chronicle* (Ghallab et al., 2004). The definition of timeline provides a unified view for both resources and logical facts.

Note that the points 3) and 4) of consistency requirements on a logical timeline are more restrictive than we can find in (Ghallab et al., 2004). The requirement of total ordering of transitions is the representation choice in FAPE, which becomes later helpful for capturing the causality. It can be seen as a simplification within the same concept as the chaining of temporal databases in Filuta (Dvořák, 2009).

In the next section we shall introduce actions as collections of temporal statements and methods to decompose them.

4.2.4 Actions and Methods

In classical planning we had actions with sets of preconditions and positive and negative effects. We knew that the preconditions must hold before the action is applied and that the effects hold after applying the action. In temporal planning we further generalize the notions of preconditions and effects. Neither of those are further restricted to the start or the end of the action but they temporarily relate to the start and the end of the action. For example, an action can have a condition that something holds a few seconds after it starts and something changes a few seconds before it ends. Such events are represented by the temporal statements (Definition 4.2.3). We shall now define the actions formally.

Definition 4.2.5. *For sets of objects O, object variables K and state variables F^i, we define an action $\alpha = (name, P, S, t_s, t_e, C)$, where:*

- *name represents the an identifier of the action, as declared in ANML.*
- *$P \subseteq K$ is a set of object variables, also called parameters of the action α,*
- *S is a set of temporal statements by the Definition 4.2.3,*
- *t_s and t_e are time points representing the start and the end of the action,*
- *C is a set of object constraints between the object variables contained in S and P, and temporal constraints between timepoints t_s, t_e and time points contained in S. For the set of temporal constraints in C there exists a temporal network $N = (V,E)$ such that the constraints are consistent with regard to N. There exists an instantiation ζ such that all constraints in C are satisfied. In other words, an action cannot contain contradictory constraints such as $x, y \in K, x = y \wedge x \neq y$, or $t_s = t_e - 10 \wedge t_s = t_e - 5$.*

We denote the set of all actions as A.

An action has time points to represent its start and end, it has some object variables as parameters and it contains a set of statements. The constraints in the action then determine the temporal relations between the statements and the start and end of the action, while the object constraint bind together the object variables contained in the statements. We can imagine an example of an action `move(Car c, Location a, Location b)` that contains statements `[all] c.location == a :-> b` and `[all] b.hasGasStation == true`. The binding constraints are represented by sharing the same variable name, we can also notice that a value used in the transition `a :-> b` becomes bound with a parameter of a parameterized state variable `b.hasGasStation`. The temporal constraints then consist of a duration of the action, e.g. `end = start + 10`, and the constraints that fixate the start and end of both statements to the start and end of the action.

Actions can be further decomposed into other actions by methods in the same fashion as in HTN planning (Definition 1.4.2). We define the methods as follows.

Definition 4.2.6. *For sets of objects O, object variables K, state variables F^i and actions A, we define a method $m = (\alpha, D, C)$, where*

- *$\alpha \in A$ is an action,*
- *$D \subseteq A$ is a set of actions, and*
- *C is a set of object and temporal constraints. For the set of temporal constraints in C there exists a temporal network $N = (V,E)$ such that the constraints are consistent with regard to N. For all object constraints in C there exists an instantiation ζ such that all the constraints are satisfied. In other words, a method cannot contain contradictory constraints.*

We denote the set of all methods as M.

The object constraints in the methods bind together parameters across actions in the decomposition, and the temporal constraints temporarily relate the start and end times of those actions. For example, a method can decompose an action `transport(Car c, Item i, Location a, Location b)` into actions `pick(Car c, Item i, Location a)`, `move(Car c, Location a, Location b)` and `drop(Car c, Item i, Location b)`, binding together their parameters and introducing temporal constraints that order the actions into a sequence. The statements in an action and the actions in its decomposition are not duplicitous. They represent a more detailed realization of the action. E.g. the action `transport` does not contain a transition statement that changes the location of the car (since it is changed in the action `move`).

Although ANML specifies how the actions should be decomposed as a part of the actions, we find the decoupled representation into actions and methods clearer, following on the representation in HTN planning Section 1.4.

In the following section we shall define the partial plan and how the actions and methods can be applied to it.

4.2.5 Plan and Refinements

The partial plans in plan space planning consist of collections of actions, and various constraints among them. Such plans may not directly correspond to a

state of the world (as in state space planning), but they record only a partial evolution of the state space. We represent the partial plan as a collection of actions and their decompositions, we maintain the temporal constraints in a temporal network, we keep the temporal statements spread across a set of timelines and we keep a set of object constraints. We shall now define the partial plan formally.

Definition 4.2.7. *For sets of objects O, object variables K, state variables F^i, actions A and methods M, partial plan is a quadruple $\pi = (N, \Phi, \mathcal{N}, C)$, where:*

- *$N = (V, E)$ is a simple temporal network.*
- *Φ is a set of timelines.*
- *$\mathcal{N}$ is a finite set of action nodes, where each node is a pair $n = (\alpha, D_\alpha)$ and D_α is a set of nodes. A node $(\alpha, \emptyset)$ is called a leaf. We say a leaf $n = (\alpha, \emptyset)$ is terminal iff $\forall(\alpha_i, D, C) \in M : \alpha_i \neq \alpha$. Further, we assume the decomposition trees are legal:*
  - *$\forall(\alpha, D_\alpha) \in \mathcal{N}, D_\alpha \neq \emptyset : D_\alpha \subset \mathcal{N}$,*
  - *$\forall(\alpha, \{(\alpha_1, D_{\alpha_1}), .., (\alpha_k, D_{\alpha_k})\}) \in \mathcal{N}, \exists(m, D, C) \in M : D = \{\alpha_1, .., \alpha_k\}$.*
- *C is a set of object constraints.*

We say a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ is consistent iff:

- *All leaves in $\mathcal{N}$ are terminal.*
- *There exists an instantiation of object variables ζ and an instantiation f_{STN} of temporal network N, such that all timelines in Φ are consistent with regard to C and N, and for every function $f^i \in F^i$, there exists at most one timeline in Φ that represents it.*
- *All logical timelines in Φ are supported.*

We denote the set of all partial plans as Π.

In contrast with the HTN planning, the task network $\mathcal{N}$ contains not only the layer of leaves, as in HTN, but also the complete decomposition trees. Intuitively, if $M = \emptyset$ then $\mathcal{N}$ becomes a flat structure of roots that are also leaves. Also, there can be multiple methods for a single action and $\mathcal{N}$ then naturally records which methods we have chosen. The legality of the decomposition trees guarantees that the a non-empty decomposition at each node corresponds to an application of some method.

We shall now describe how the actions are inserted into a partial plan and how they are further decomposed. It is important to note that since a single action can occur multiple times in a partial plan, the insertion of an action α substitutes the object variables in α with a set of new object variables with the same domains. In the same way, we create new time points and consider all the object and temporal constraints to be mapped on the new variables. Therefore, two insertions of a single action are standardized apart. This principle is also often captured by distinguishing *operators*, as the abstract schemes of actions, and actual actions in the plan as the *instances* of those operators. We shall describe how the actions are added to a plan and how they are further decomposed.

Action Insertion
An action is a collection of time points, statements, parameters and constraints. When we insert a new action into a partial plan, the temporal constraints consist of the relations between the statements of the action and the start and the end of the action. The object constraints then represent binding of object variables, used as parameters of the action, with the object variables used in the statements. In ANML terminology, having an action

```
move(Robot r, Location l1, Location l2)
```

and its statement `r.location == l1 :-> l2`, the object variables annotated by the same symbol, such as `r,l1,l2`, become bound.

We denote the action insertion as a function $\xi : A \times \Pi \to \Pi$. For an action $\alpha = (name, P, S, t_s, t_e, C_\alpha)$ and a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$, the insertion of α into π is defined as

$$\xi(\alpha, \pi) = (N', \Phi', \mathcal{N}', C'), \text{ where:}$$

- $N' = (V', E')$ is constructed from $N = (V, E)$, by adding action time points, $V' = V \cup \{t_s, t_e\}$ and adding duration constraint $(t_e = t_s + duration) \in C_\alpha$. Further, all time points contained in statements S are added to V', and all temporal constraints in C_α are added into E. The time points contained in statements can be the time points t_s and t_e, e.g. when the statements relate directly to the action's start and end, `[start,end]` in ANML, and also new intermediate time points, representing annotations such as `[start+2, end-1]` and `[100]`. For each such temporal annotation a new time point is created in V' and temporal constraints that represent the temporal annotation are added into E. For example, `[start+2, end-1]` becomes represented as two new time points t_1 and t_2, and constraints $t_1 = t_s + 2$ and $t_2 = t_e - 1$.
- $\Phi' = \Phi \cup \{\{x\}, x \in S\}$. In other words, we create a new timeline for each statement in S.
- $\mathcal{N}' = \mathcal{N} \cup (\alpha, \emptyset)$. In other words, we add a new root to the task network.
- C' is constructed from C by adding all the object constraints in C_α. Those are the object constraints that bind together variables in statements with the variables in the parameters P of the action α.

Note that the action insertion by itself does not relate or constrain it to other timelines and time points in a partial plan. However, an action insertion is always motivated by some inconsistency of the plan, and the context of such inconsistency determines how the action shall connect. We go into detail later in this section.

Action Decomposition
A method consists of an action to decompose, a set of actions which it decomposes into, object constraints that bind together parameters of all actions and temporal constraints that relate the actions in the decompositions. An example would be an action `travel(Person p, Location l)`, where having actions `pick(Person x, Location l1)`, `drive(Location l1, Location l2)` in

the decomposition binds together `x = p`, `l2 = l`, and adds a precedence constraint `pick` $<$ `drive`.

We denote a decomposition of an action through a method as a function $\psi : A \times M \times \Pi \rightarrow \Pi$. For a method $m = (\alpha_m, D, C_m)$, $D = \{\alpha_1, .., \alpha_n\}$ and a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$, such that $(\alpha, \emptyset) \in \mathcal{N}$ and α_m shares the name with α, we define the decomposition of α in π as

$$\psi(\alpha, m, \pi) = (N', \Phi', \mathcal{N}', C')$$

A decomposition is constructed in three steps:

1. We first insert all the actions using the action insertion function ξ as described for the action insertion, producing a new plan π',

   $$\pi' = \xi(\alpha_n, .., \xi(\alpha_1, \pi)).$$

   Action insertion ξ creates a new leaf for each inserted action. We would like to record the actions of decomposition in the node of the action that has been decomposed. Having $\pi' = (N', \Phi', \mathcal{N}', C')$ as an output of ξ, we update the original leaf node to $(\alpha, \{(\alpha_1, \emptyset), .., (\alpha_n, \emptyset)\})$. The action insertion ξ already applied all the constraints inside the actions and we are left with adding the constraints in the method. An example of several decompositions of actions is illustrated in Figure 4.1.

2. We update C' with the object constraints C_m in the method, which consist of constraints between the parameters of the action α and parameters of actions $\alpha_1, .., \alpha_n$.

3. Having $N' = (V', E')$, we update E' with all temporal constraints in C_m.

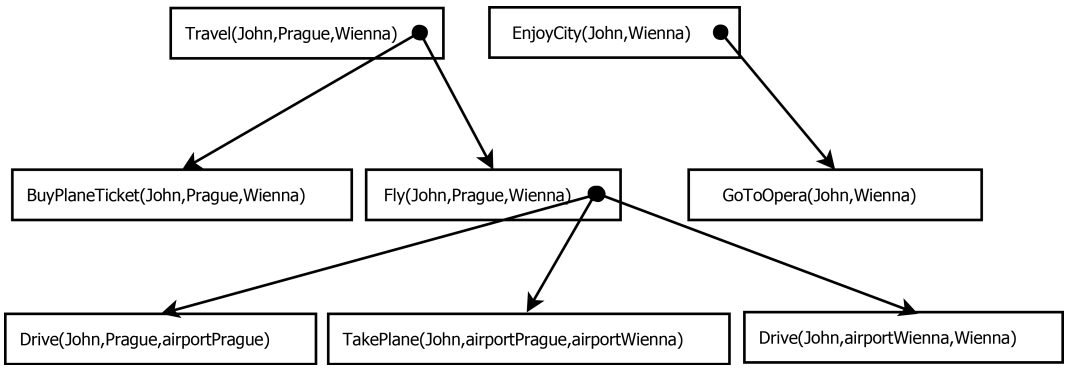


Figure 4.1: Figure shows an example how the task network structure is organized. An action is represented exactly once, a leaf can be further decomposed if there exists a method, whose action unifies with the action in the leaf.

The action decomposition and insertion are the key steps through which we progress in planning in FAPE. However, they are not the only one, since the representation contains other decision points that need to be dealt with before a partial plan becomes consistent. We follow on the plan space planning Section 1.3 and adopt the approach of describing the issues of the partial plan through *flaws* and their *resolvers*.

4.2.6 Flaws and Resolvers

We are planning in a plan space, where different search states represent partial plans. A partial plan consists of logical timelines that may be unsupported, resource timelines that may be inconsistent and a task network, where we have not yet decomposed all the actions. We call such plan deficiencies to be *flaws* of the partial plan. A flaw maybe resolved by performing some set of operations, such as joining timelines and adding actions. We call such sets of operations *resolvers* of the flaw. In this section we shall describe the flaws we identify and the resolvers we construct for them.

For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$, we identify four types of flaws:

- *Unsupported logical timeline.* In an unsupported logical timeline ϕ, there is either no transition statement, or the earliest transition statement is not an assignment statement.

- *Inconsistent resource timeline.* The corresponding resource temporal network is inconsistent in terms of Definition 3.1.2.

- *Undecomposed action.* A non-terminal leaf node in a network $\mathcal{N}$ is a flaw.

- *Threat.* There exist two different logical timelines $\phi_i, \phi_j \in \Phi, i \neq j$ such that ϕ_i unifies with ϕ_j and they may possibly intersect (the earliest statement of ϕ_i may happen before the latest statement in ϕ_j and the earliest statement of ϕ_j may happen before the latest statement in ϕ_i with regard to N).

- *Unbound timeline.* A timeline represents an evolution of some state variable and we may not know which ground state variable until the object variables become bound. We denote a set of all possible state variables a timeline ϕ can represent as:

$$F^i_\phi = \{f^i | \forall s \in \phi : f^i \; is \; instance \; of \; var(s)\}.$$

  If $|F^i_\phi| > 1$, then ϕ forms a flaw of a plan.

- *State Variable Competition.* Two different timelines ϕ_i and ϕ_j form a flaw if, $|F^i_\phi| = 1$, $|F^i_{\phi_i}| = 1$, and $F_{\phi_i} = F_\phi$. In other words, two timelines that necessarily represent the same state variable are a flaw of the plan.

The unsupported logical timeline is a flaw that we can resolve by finding some other timeline, where the latest transition provides a value that can support it. Such timeline may already exist in the partial plan, we may also insert an action that produces such supporting timeline, either directly, or through decomposition. The process of supporting a timeline can be seen as an addition of a causal link (see Section 1.3) since the latest transition, temporal statement, in the supporting timeline can be seen as an effect of an action that provided the statement. Having an unsupported timeline and a timeline that can support it, we merge (concatenate with regard to transitions) both timelines. We shall now describe the construction of such resolvers in detail, starting with a definition of a *supporting timeline.*

Definition 4.2.8. *For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and an unsupported consistent logical timeline $\phi \in \Phi$, we say that a consistent timeline $\phi_j \in \Phi$, with at least a single transition, supports ϕ, iff the following conditions are satisfied:*

- *ϕ unifies with ϕ_j,*
- *If ϕ consists of a single persistent condition $P(f, t_s, t_e, u)$ then there exists a transition $x_i = T(f', t'_s, t'_e, u', v') \in \phi_j$ such that:*
  - *$t'_e \leq t_s$ is consistent with N,*
  - *$u = v'$ is consistent with C, and*
  - *If x_i is not the latest transition in ϕ_j then after x_i there exists the earliest transition $x_{i+1} = T(f'', t''_s, t''_e, u'', v'')$ such that $u = u''$ is consistent with C and $t'_e \leq t_s \wedge t_e \leq t''_s$ is consistent with N.*
- *If ϕ contains an earliest transition transition $T(f, t_s, t_e, u, v)$, ϕ_j contains a latest transition $T(f', t'_s, t'_e, u', v')$, and*
  - *$t'_e \leq t_s$ is consistent with N,*
  - *$u = v'$ is consistent with C,*
  - *For a set of constraints*

$$E_\phi = \{t'_e \leq t_a | P(f, t_a, t_b, w) \in \phi, t_b > t_s \text{ is inconsistent with } N\}$$
$$E_{\phi_j} = \{t_b \leq t_s | P(f, t_a, t_b, w) \in \phi_j, t_a < t'_e \text{ is inconsistent with } N\}$$

  $E_\phi \cup E_{\phi_j}$ is consistent with N.

The definition says that a timeline ϕ can be supported by another timeline ϕ_j if they can be temporarily concatenated, sharing the values on the edges and leaving enough time for all persistent conditions on the edges to fit between two transitions. A timeline with a single persistent condition is a special case, which can get merged between two transitions of another timeline, if the object variables can bind and the time window is large enough. Note that there can never exist a timeline consisting solely of two persistent conditions, since we shall only merge timelines when one can support another and the supporting timeline needs to contain at least one transition to provide such support. Figure 4.2 illustrates an example of a timeline supporting another one. Further, in the last point of the definition, the consistency of $u = v'$ with C ensures that also all the persistant conditions in between the two transitions share the value, since they have been bound already to either u or v'.

Resolvers for Unsupported Logical Timelines

We can now describe resolvers for an unsupported timeline. For convenience, we assume there exists a function $Refine : R \times \Pi \to \Pi$ that applies a resolver to an input partial plan as we further describe and produces a new partial plan.

For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and an unsupported consistent logical timeline $\phi \in \Phi$, we suggest three resolvers:

- *Merge.* Having a timeline $\phi_j \in \Phi$ that supports ϕ we construct a resolver $Merge(\phi, \phi_j, s)$, where $s \in \mathcal{T} \cap \phi_j$, as follows:

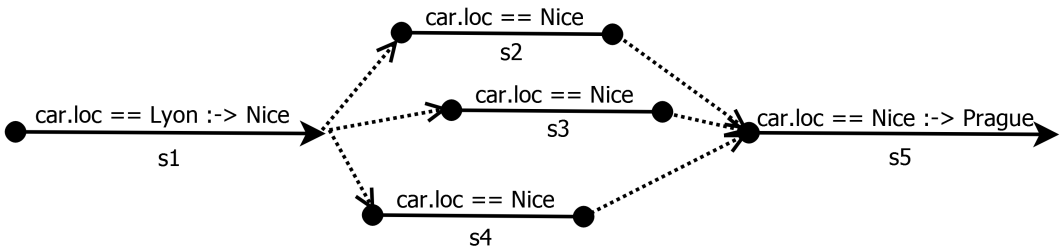


Figure 4.2: Figure shows two timelines $\phi = \{s4, s5\}$ and $\phi_i = \{s1, s2, s3\}$, where ϕ_i supports ϕ, and how they can be merged. Dotted lines represent the temporal precedence constraints.

- An assignment $\phi_j \leftarrow \phi_j \cup \phi$ and binding the object variables used as parameters of their parameterized state variables.
- A removal of ϕ from Φ.
- If ϕ consists of a single persistent condition $P(f, t_s, t_e, u)$, then each transition $s_i \in \phi_j$ that supports the condition in terms of Definition 4.2.8 forms an instance of this resolver (determines the statement s_i in $Merge(\phi, \phi_j, s_i)$). Then having such a transition $s_i = T(f', t'_s, t'_e, u', v')$, we enforce constraints $u = v'$ into C and $t'_e \leq t_s$ into N. If there exists a transition $s_{i+1} = T(f'', t''_s, t''_e, u'', v'')$ after s_i, we further enforce $u = u''$ into C and $t_e \leq t''_s$ into N.
- If ϕ contains an earliest transition transition $T(f, t_s, t_e, u, v)$, ϕ_j contains a latest transition $s = T(f', t'_s, t'_e, u', v')$, then
  - we enforce $t'_e \leq t_s$ into N,
  - we enforce $u = v'$ into C,
  - $\forall P(f, t_a, t_b, w) \in \phi : t_b > t_s$ is inconsistent with N then we enforce $t'_e \leq t_a$ into N, and
  - $\forall P(f, t_a, t_b, w) \in \phi_j : t_a < t'_e$ is inconsistent with N then we enforce $t_b \leq t_s$ into N.

  The construction of these resolvers follows the Definition 4.2.8, we introduce only those constraints that are already known to be individually consistent.

- *Action Insertion.* Having an action $\alpha = (name, P, S, t_s, t_e, C_\alpha)$ such that there exists a transition statement $s = T(f, t_s, t_e, u, v) \in S$ and a timeline $\{s\}$ would support ϕ in terms of Definition 4.2.8, we construct a resolver $Action(\phi, \alpha, s)$ in two steps:
  - We first apply the action α as $\pi' = \xi(\alpha, \pi)$.
  - Then we apply the merge resolver for the new supporting statements s introduced by the action α as $\pi'' = Refine(\pi', Merge(\phi, \{s\}, s))$.

  An action insertion can be a resolver of an unsupported logical timeline if it contains a statement that supports the timeline. The unsupported timeline also represents the only temporal relation for the inserted action and the object binding occurs only on the object variables contained in

the supported timeline. The other statements introduced by the action insertion form new timelines.

The unsupported timeline corresponds to an unsupported precondition in the plan space planning (Section 1.3), and the resolvers are concepts how to add a causal link to support it. We shall now show, that the resolvers produce only consistent logical timelines.

Lemma 4.2.9. *For a set of actions A, a set of methods M and a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$, where N is consistent and all logical timelines Φ are consistent with respect to N and C, all logical timelines produced by resolvers Merge and Action are also consistent.*

Proof. A timeline ϕ is consistent if the conditions 1) - 4) in Definition 4.2.4 are satisfied. We shall first show that $Merge(\phi, \phi_j, s)$ resolver produces a consistent timeline. Without the loss of generality, we assume there exists an unsupported consistent logical timeline $\phi \in \Phi$ and a consistent logical timeline $\phi_j \in \Phi$ that supports ϕ. We shall show that $\phi_j \cup \phi$ is consistent.

We start with a case when ϕ consists of a single persistent condition $P(f, t_s, t_e, u)$ and there exists a transition $s = x_i = T(f', t'_s, t'_e, u', v') \in \phi_j$. Since ϕ_j supports ϕ in terms of Definition 4.2.8, we know that $C \cup \{v' = u\}$ is consistent and $N = (V, E \cup \{t'_e \leq t_s\})$ is consistent therefore there exists a consistent instantiation of object variables ζ and instantiation f_{STN} of N, satisfying the point 1). The same holds for a transition statement x_{i+1} just after x_i, if there such a statement exists; we further do not consider x_{i+1}, since all the reasoning is symmetrical to x_i. Further, we know that ϕ_j is consistent, therefore the point 2) could be invalidated only by a combination of a persistent condition $P(f'', t''_s, t''_e, u'') \in \phi_j$ and $P(f, t_s, t_e, u)$. We assume the two persistent conditions intersect (otherwise they satisfy condition 2)), $t''_s < t_e$ and $t_s < t''_e$. By point 4) $x_i = T(f', t'_s, t'_e, u', v')$ is the latest transition before $P(f'', t''_s, t''_e, u'')$, therefore $v' = u''$ is consistent with C. Since $v' = u''$ and $v' = u$ are consistent, $u = u''$ must be transitively also consistent satisfying the point 2). Condition in point 3) is not effected by adding a persistent condition and we have already added the constraints to satisfy point 4).

In second case, ϕ contains the earliest transition $T(f, t_s, t_e, u, v)$ and there exists the latest transition $T(f', t'_s, t'_e, u', v') \in \phi_j$. Further, $u = v'$ and $t_e \leq t'_s$ are consistent by Definition 4.2.8 and their introduction satisfies the total ordering in point 3). We denote all the persistent conditions in ϕ that are ordered before the earliest transition as P_ϕ, and all the persistent conditions in ϕ_j that are ordered after the latest transition as P_{ϕ_j}. By Definition 4.2.8 we know, that all the statements P_ϕ can be ordered after $T(f', t'_s, t'_e, u', v')$ and all the statements in P_{ϕ_j} can be ordered before $T(f, t_s, t_e, u, v)$, therefore by introducing the ordering constraints for the two sets, the persistent conditions in $P_{\phi_j} \cup P_\phi$ become ordered in between transitions $s = T(f', t'_s, t'_e, u', v')$ and $T(f, t_s, t_e, u, v)$, satisfying the point 4). Consequently, $u = v'$ is consistent with C and since ϕ is consistent, we know that $\forall P(f_x, t_{sx}, t_{ex}, w_x) \in P_\phi : w_x = u$ is consistent with C, and since ϕ_j is consistent $\forall P(f_y, t_{sy}, t_{ey}, w_y) \in P_{\phi_j} : w_y = v'$. From transitivity of binding we get that $\forall P(f_x, t_{sx}, t_{ex}, w_x) \in P_\phi, \forall P(f_y, t_{sy}, t_{ey}, w_y) \in P_{\phi_j} : w_x = w_y$ is consistent, satisfying the points 2) and 4).

We have showed that the application of $Merge(\phi, \phi_j, s)$ does not make logical timelines inconsistent. Further, resolver $Action(\alpha, s)$ creates new timelines corresponding to its statements. Since a timeline consisting of a single statement is always consistent according to Definition 4.2.4, and all the temporal and object variables in α are standardized apart, without introducing contradictory constraints by Definition 4.2.5, the logical timelines introduced by α are consistent. Applying $Merge$ as a part of $Action$ maintains the consistency of the logical timelines.

□

Note that while only consistent timelines are produced when inserting an action, the other statements produced by an action may require to be supported as well, forming new flaws.

Resolvers for Inconsistent Resource Timelines
For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and an inconsistent resource timeline $\phi \in \Phi$, we construct following resolvers:

- *Merge.* Having a resource timeline $\phi_i \in F$ that unifies with ϕ and either $\phi \cup \phi_i$ is a consistent resource or ϕ_i *contributes* to resolving a flaw in ϕ, we construct the resolver $Merge(\phi, \phi_i)$ as follows:

  - An assignment $\phi_i \leftarrow \phi_i \cup \phi$ and binding the object variables used as parameters of their parameterized state variables.
  - A removal of ϕ from Φ.

  The resolver represents merging of temporal developments of a resource, the merge can be invoked as a resolver to an inconsistency of a *reservoir* resource, determined by the *balance constraints* Section 3.2.1. We say that ϕ_i contributes to resolving a flaw in ϕ, if ϕ_i consists of a single statement, such that the resource overconsumption or overproduction reduces.

- *Action Insertion.* Action can be applied to resolve an inconsistent reservoir resource, the balance constraint produces an information when the conflict occurs, determined by time point t_{bc} and whether the resource is overconsumed or overproduced. Having an action $\alpha = (name, P, S, t_s, t_e, C_\alpha)$ such that there exists $R(f, t, x) \in S$, $x < 0$ if the resource is overproduced, or $x > 0$ if the resource is overconsumed, and $\{R(f, t, x)\}$ unifies with ϕ, the resolver $Action(\phi, \alpha, s, t_{bc})$ is constructed as follows:

  - We first apply the action α as $\pi' = \xi(\alpha, \pi)$.
  - We introduce a new precedence constraint $t \leq t_{bc}$ into temporal network in π'.
  - Then we apply the merge resolver for the new statements s introduced by the action α as $\pi'' = Refine(\pi', Merge(\phi, \{s\}))$.

- *Temporal Constraint.* Precedence constraints are generated for both discrete resources and reservoirs (see Chapter 3). They have a form $t_x \leq t_y$ and they are known to be individually consistent with the temporal network

N, since the balance constraints and minimal critical sets produce only consistent precedence constraints. Intuitively, different precedence constraints may be contradictory and the choice of the resolver shapes the plan.

Resolvers for Threats and Undecomposed Actions
For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and a threat formed by $\phi_i, \phi_j \in \Phi, i \neq j$, we formulate two types of resolvers:

- Precedence Constraints. Either the latest statement of ϕ_i must occur before the earliest statement in ϕ_j with regard to N, or vice-versa.
- Object Constraints. The resolver is each such object variable separation constraint $x \neq y$ that ϕ_i does not longer unify with ϕ_j.

The concept of the threat is to identify those pairs of timelines that would threaten each other by breaking the causality relations they capture. We can imagine an example of having a timeline {`[0,100] x.location == apartment`} and {`[50,200] y.location == bridge`}, then the only resolver for such threat is a separation constraint `x` $\neq$ `y`. The concept of the threat is quite close to the one in plan space planning Section 1.3, but since we encode the concept of causal links through the total ordering of transition statements on timelines, we also define the threats on the level of timelines.

For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and a flaw of an undecomposed action α and each method $m \in M, m = (\alpha, D, C_m)$, such that C_m is consistent when inserted through $\psi(\alpha, m, \pi)$, we construct a resolver $Decompose(\alpha, m)$.

Resolvers for Unbound and Competing Timelines
For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and a unbound timeline ϕ a resolver is each set of assignments of values to the object variables in ϕ, such that $|F^i_\phi| = 1$. In other words, the resolver grounds the parameterized state variables in ϕ. We can imagine an example of a timeline

```
{r.position == bridge :-> apartment,
 r.position == apartment,
 r.position == apartment :-> airport}
```

where `r` $\in$ `{r1,r2}`. Then an assignment `r` $\leftarrow$ `r1` is a resolver of the flaw.

Note that the purpose of identifying this flaw is mainly technical, in reasonable planning problems, the parameterized state variables become ground through propagation of binding constraints during planning. E.g., the initialization of a state variable value represents a support for such unbound timeline and as such it grounds it when they merge.

For a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ and competing timelines ϕ_i and ϕ_j, $Merge(\phi_i, \phi_j)$ and $Merge(\phi_j, \phi_i)$ are resolvers. The notion is intuitive, if we are sure that two timelines represent the same state variable, we need to merge them. Independently whether they are resource timelines or logical timelines, we use the corresponding $Merge$ resolver as defined in previous sections. Again, the flaw is technical and during planning, we implicitly merge all competing resource timelines and let the competing logic timelines disappear through resolvers of other flaws.

For a partial plan π, we say the applications of resolvers to the flaws in π are *refinements* of π. We have now set up all the pieces needed for defining a planning problem in the next section.

4.2.7 Planning Problem

A planning problem consists of a sets of objects, methods, actions and some initial partial plan π_0. In classical planning (Definition 1.0.1) we have specified the goals explicitly as the atoms that need to be achieved. Then in HTN planning (Definition 1.4.2) the goal was to perform a set of tasks by decomposing them into primitive tasks. We merge both of those approaches by representing the goals as unsupported timelines in π, e.g. `[end]item.location == destination`, and the task to decompose as the undecomposed actions in π, e.g. by having (`transport(item,anywhere,destination)`, $\emptyset$) in the task network. Further, by integrating resources in the form of resource timelines we may as well represent goals upon them, e.g. if we need to produce a thousand of bricks, we add an initial consumption statement `bricks :consume 1000` into π as a resource timeline. Using this concepts, the achievement of goals is equivalent to finding a consistent plan π_g by refining π. We shall now define the planning problem formally.

Definition 4.2.10. *The planning problem is a tuple $P = (\pi_0, O, M, A)$, where π_0 is the initial partial plan, where all logical timelines are consistent and all statements contain only ground state variables, O is a set of objects, M is a set of methods and A is a set of actions.*

A partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ is the solution of the planning problem $P = (\pi_0, O, M, A)$, $\pi_0 = (N_0, \Phi_0, \mathcal{N}_0, C_0)$, if the following conditions are satisfied:

1. *π is a consistent plan.*
2. *$C_0 \subseteq C$, in other words, all the initial constraints are represented in C.*
3. *For $N_0 = (V_0, E_0)$ and $N = (V, E)$, $V_0 \subseteq V$ and $(V, E \cup E_0)$ is consistent. In other words, all the initial time points remain in the plan and all the initial temporal constraints are satisfied.*
4. *$\forall \phi_0 \in \Phi_0, \exists \phi \in \Phi : \phi_0 \subseteq \phi$. In other words, all the initial developments of the state variables are represented in Φ.*
5. *$\forall \phi_i, \phi_j \in \Phi_0$, such that all state variables in $\phi_j \cup \phi_i$ represent the same ground state variable, $\exists \phi \in \Phi : \phi_j \cup \phi_i \subseteq \phi$. In other words, when we set up two statements for a state variable in the initial plan, they must be contained in a single timeline in the solution. E.g., having*

   ```
   [end] r1.location == apartment
   [start] r1.location := bridge
   ```

   in the initial plan, then these statements must be combined into a single timeline such as

   ```
   {r1.location := bridge,
    r1.location == bridge :-> apartment,
    r1.location == apartment}
   ```

6. *Having a set of all statements in all actions contained in* $\mathcal{N}$*,*

$$S_{\mathcal{N}} = \{s \in S|, (\alpha, D_\alpha) \in \mathcal{N}, \alpha = (name, P, S, t_s, t_e, C)\}$$

then $S_{\mathcal{N}} \cup \{s \in \phi | \phi \in \Phi_0\} = \{s \in \phi | \phi \in \Phi\}$*. In other words, all the statements that form the timelines of the solution plan must either come from an action in the task network, or be a part of the initial partial plan.*

The planning problem $P = (\pi_0, O, M, A)$ is at least EXPSPACE-hard, since it subsumes the problem of concurrent temporal planning (Rintanen, 2007).

Lemma 4.2.11. *For a problem* $P = (\pi_0, O, M, A)$*, a partial plan* π *obtained by refining* π_0 *and having no flaws is a solution.*

Proof. We shall first show that the plan is consistent with regard to Definition 4.2.7. Having no flaws of undecomposed actions, then all nodes in $\mathcal{N}$ must be terminal. Further, if there are no flaws of unsupported logical timelines, then all logical timelines must be supported. If there are no flaws of inconsistent resource timelines, then all resource timelines are consistent. Since π_0 has all logical timelines consistent and the refinements through resolvers retain the consistency of logical timelines by Lemma 4.2.9, all the logical timelines in π are consistent. Finally, since there are no flaws of unbound and competing timelines, every state variable in F^i is represented at most once. We have shown that π is consistent.

The refinements of a plan through resolvers never remove a time point, temporal constraint or an object constraint. Therefore the second and third point of Definition 4.2.10 are satisfied. Further, timelines can only be merged through resolvers, but they are never split, satisfying the fourth point. Since there are no flaws of competing timelines, any pair of timelines, whose parameterized state variables represent a single ground state variable, must have been necessarily merged together, satisfying the fifth point.

Since we only insert and decompose actions, sixth point is trivially satisfied. □

We have shown that if we can reach a partial plan with no flaws by resolving the flaws, it is a solution. Hence an algorithm, that refines flaws until there are none, is correct.

Further, we can observe that any task network, with legal decomposition trees, of any partial plan can be constructed by adding a set of actions s_α and then applying a sequence of methods s_m. We shall show that if there exists a solution to a problem, where either $s_\alpha = \emptyset$, or $s_m = \emptyset$, then we can find a plan by refining the initial plan. Then we give an example that even if there exists a plan, where $s_\alpha \neq \emptyset \wedge s_m \neq \emptyset$, the refinements of the initial plan may not lead to a solution. We then propose a new resolver, and give an intuition how to prove that if there exists a plan, we can find a plan through refinements.

Theorem 4.2.12. *If there exists a solution plan* $\pi = (N, \Phi, \mathcal{N}, C)$ *to a planning problem* $P = (\pi_0, O, M, A)$*, such that the set of inserted actions* $s_\alpha = \emptyset$*, then there exists a solution plan* $\pi' = (N', \Phi', \mathcal{N}', C')$ *produced by refining* $\pi_0 = (N_0, \Phi_0, \mathcal{N}_0, C_0)$*.*

Proof. We shall construct a sequence of refinements s_r that leads from the initial plan π_0 to a solution π'. We shall show how a sequence of resolvers is generated, using the plan π' as an oracular that determines the choices of resolvers.

We initialize $s_r = ()$, $\pi^* = \pi_0$ and repeat the following steps.

1. If π^* has a flaw of competing resource timelines $\phi_i, \phi_j \in \Phi^*$, then $r \leftarrow Merge(\phi_i, \phi_j)$ and go to 8).

2. If π^* has a flaw of undecomposed action α, and there exists $(m_i, \alpha) \in s_m$, then $r \leftarrow Decompose(m_i, \alpha)$ and go to 8).

3. If π^* has a flaw of an unsupported timeline $\phi^* \in \Phi^*$, ϕ^* either consists of a single persistant conditions s_i or it has an earliest transition s_i, then $\exists s_{i-1} \in \phi \in \Phi : s_i \in \phi$ and s_{i-1} is the latest transition before s_i, $\exists \phi \in \Phi^*, s_{i-1} \in \phi$, $r \leftarrow Merge(\phi^*, \phi)$, and go to 8).

4. If π^* has a flaw of an inconsistent resource timeline (reservoir) $\phi^* \in \Phi^*$, then $\exists s \in \phi \in \Phi, \phi^* \subseteq \phi$, $\exists \phi \in \Phi^*, s \in \phi$, then $r \leftarrow Merge(\phi^*, \phi)$, and go to 8).

5. If π^* has a flaw of an inconsistent resource timeline $\phi^* \in \Phi^*$ and the resource reasoning provides a resolver $(t_x \leq t_y)$ consistent with N, then $r \leftarrow (t_x \leq t_y)$, and go to 8).

6. If π^* has a flaw of an unbound timeline $\phi^* \in \Phi^*$, then $\exists \phi \in \Phi, \phi^* \subseteq \phi$, $f(x_1, .., x_n) = var(s), s \in \phi^*, f(y_1, .., y_n) = var(s'), s' \in \phi$, we construct a set of unary constraints $S = \{(dom(x_i) \subseteq dom(y_i)), i \in \{1, .., n\}\}$, $r \leftarrow S$, and go to 8).

7. Terminate.

8. $s_r \leftarrow s_r.(r)$, $\pi^* \leftarrow Refine(\pi^*, r)$.

At each step 1) – 6) of the iteration we choose a resolver of a flaw that can be applied to the partial plan π^*, then apply it to π^* and push the resolver into the sequence s_r. Note that this implies, that all the actions are decomposed at step 2) before reaching flaws at step 3) and 4). Further, steps 3) and 4) merge timelines in the same way, as they are constructed in Φ. Since all methods were applied, then $\mathcal{N} = \mathcal{N}^*$ and in turn, all statements of actions in $\mathcal{N}^*$ are in Φ. Therefore at steps 3) and 4), there must always exist a resolver.

Since there exists a finite number of action insertions and decompositions in π by Definition 4.2.7, the number of timelines in Φ is finite. Further, the only inserted actions come from decompositions by methods s_m, therefore only a finite number of new timelines can be introduced, and steps 1), 2), 3), 4) and 6) can be applied only finite number of times. Balance constraints and minimal critical sets can produce only a finite set of temporal constraints. As a result, there is finite number of iterations, eventually reaching step 7) and terminating.

We have applied all the methods in s_m, hence $\mathcal{N}^* = \mathcal{N}$. Since all statements in timelines in Φ are either in the initial timelines Φ_0 or they are produced by an action, then $\Phi^* = \Phi$. All the object constraints in C^* and temporal constraints in E^* are either in the initial constraints C_0 and E_0, or they are produced by action

insertions, decompositions or step 5). In each case, those constraints must have been satisfied in C and E. Hence given the consistence of π, C^* and $N^* = (V^*, E^*)$ must be also consistent, and in turn, $\pi*$ is consistent.

If π^* had a flaw of an undecomposed action, then such action would appear in π and such flaw must have been resolved by step 2). The only flaws that may remain in π^* are the threats and unbound timelines. Since π is consistent, there exists a instantiation of object variables ζ, such that all timelines in Φ are consistent and each represents exactly one state variable $f^i \in F^i$. We construct a set of resolvers for unbound timelines that instantiate object variables according to ζ, append them to r_s and apply to π^*, producing π'. π' has no flaws and has been constructed by a sequence of refinements r_s, hence π' is a solution of the problem $P = (\pi_0, O, M, A)$ by Lemma 4.2.11. □

Theorem 4.2.13. *If there exists a solution plan $\pi = (N, \Phi, \mathcal{N}, C)$ to a planning problem $P = (\pi_0, O, M, A)$, such that the set of applied methods $s_m = \emptyset$, then there exists a solution plan $\pi' = (N', \Phi', \mathcal{N}', C')$ produced by refining $\pi_0 = (N_0, \Phi_0, \mathcal{N}_0, C_0)$.*

The proof is analogous to the proof of Theorem 4.2.12. Instead of step 2), we allow the actions to be inserted if they are in s_α and support some unsupported timeline, or inconsistent resource timeline.

If we have $P = (\pi_0, O, M, A)$ and its solution $\pi = (N, \Phi, \mathcal{N}, C)$, we may not guarantee that a plan can be found through refinements. The main issue is that we only insert actions into the plan as resolvers to resource conflicts or to provide support for a unsupported timelines. However, an action, that provides the support for some initial timeline, can be nested deeper in the decomposition tree, where the root of the tree could not be added as a refinement, since it does not contain any relevant statement. For example, we assume we have an action A in a non-root node of $\mathcal{N}$, whose statement is a part of some timeline $\phi \in \Phi$, such that there exists $\phi_0 \in \Phi_0 : \phi_0 \in \phi$. Then we denote the root of the tree, where the node A is contained, as R. We assume that R contains a statement `AcanBeApplied := true`, A contains statements

```
AcanBeApplied == true;
AwasApplied == false :-> true;
x == y :-> z;
```

where `AwasApplied := false` is contained in some initial timeline. Then A can occur only once in any plan, however if the statement `x == y :-> z` is contained in a timeline that also contains an initial timeline, then A must a be a part of any solution plan and it can only be added as a new root (nothing motivates the addition of R before A appears in a plan). Then by adding A, we get a new timeline `{AcanBeApplied == true}`, which needs to be supported, and the only support is provided by adding R as a new root. R now needs to be decomposed, eventually producing A again, which adds a `AwasApplied == false :-> true` that can never be supported, and as a result, the refining process can never eliminate all the flaws.

Following the concept of flaws and their resolvers, we can address such situations by introducing a new resolver *AddTree* for the flaw of unsupported logical timelines and inconsistent resource timelines. The resolver consists of an insertion

of an action a, which can decompose into an action b that is a *Action* resolver of such flaw, and decomposing a until b is added. Extending the resolvers with $AddTree$ would make any algorithm based on exhaustively exploring sequences of refinements complete. The intuition of the proof is the same as in for Theorem 4.2.12, where we further extend steps 3) and 4) with a resolvers *Action* and $AddTree$. However, the $AddTree$ resolver is difficult to capture, for a single flaw and recursive methods, the number of $AddTree$ resolvers can be infinite. If the methods are not recursive, than the number of $AddTree$ resolvers for a single flaw can be exponential in the number of action (we generate a set of resolvers for each path in a possible decomposition tree, where the path is a permutation of actions). As a result, generating all $AddTree$ resolvers is impractical. It can be addressed to some merit by limiting the depth of trees, or further constraining the possible decompositions, e.g. by the `motivated` statement in ANML as discussed in Future Work. And of course, a reasonable construction of planning problems can avoid the exponential generation of $AddTree$ resolvers, however, the general case remains difficult.

In the following section we approach the problem algorithmically.

4.3 Search

We have defined the planning problem and its solution that can be reached by performing a sequence of refinements to the initial partial plan, if there some solution exists. The choices of the refinements of a partial plan (resolvers of flaws) form the decision points. In Chapter 1 we have presented several standard algorithms for planning, where the Algorithm 5 is in principle solving the same abstract problem (refining until no flaws remain) as the algorithm we shall present here. All those algorithms had in common a strong dependence on the quality of heuristics that guides them. We have designed the search algorithm in FAPE to be modularly extendable with various heuristics and strategies, and we present its generic form in Algorithm 11.

The algorithm is continually improving a plan until the *best* plan is found or it is stopped arbitrarily. At lines 6-8 we check whether there are any partial plans left to refine and return *best* otherwise. At line 9 we choose the partial plan π to refine and generate the flaws of π at line 11. If there are no flaws, we update the *best* plan and prune all the inferior plans from the set q. Otherwise, we choose a flaw at line 16, generate resolvers for the flaw, and for each resolver we generate a new partial plan π', which is inserted into q unless it is a dead-end.

Procedure $Flaws(\pi)$ finds all the flaws for the given plan. $Refine(\pi, r)$ applies the resolver r as described in the previous section. Procedure $DeadEnd(\pi)$ determines whether the partial plan π can be considered a dead-end of the search by checking the consistency of the temporal network, whether there exists a solution to all the binding constraints and whether there exists a flaw with an empty set of resolvers.

The algorithm can be run in three modes, depending on the definition of the procedures $Better$ and $Prune$:

- *Continual Improvement.* This is a mode where for a given metric $\sigma : \Pi \to \mathbb{R}^+$ procedure $Better$ evaluates the partial plan π as $Better(a, b) =$

Algorithm 11 FAPE Search

Input: A planning problem $P = (\pi_0, O, M, A)$.
Output: Solution plan or a $failure$.

```
1: q ← {π0}; best ← failure
2: while true do
3:     if stopped then
4:         return best
5:     end if
6:     if q = ∅ then
7:         return best
8:     end if
9:     π ← ChoosePlan(q)
10:    q ← q \ {π}
11:    flaws ← Flaws(π)
12:    if flaws = ∅ then
13:        best ← Better(π, best)
14:        q ← Prune(q, best)
15:    else
16:        f ← ChooseFlaw(flaws)
17:        resolvers ← Resolvers(f)
18:        for all r ∈ resolvers do
19:            π' ← Refine(π, r)
20:            if not DeadEnd(π') then
21:                q ← q ∪ {π'}
22:            end if
23:        end for
24:    end if
25: end while
```

$argmin_{x \in \{a,b\}}(\sigma(x))$. The procedure $Prune$ does nothing in this settings, $Prune(q, \pi) = q$.

- *Optimization.* This mode is an extension of the continual improvement, where the given metric can be defined as $\sigma(\pi) = g(\pi) + h(\pi)$, where g is an evaluation of the cost of π and h is an admissible heuristic that computes the cost of reaching a consistent plan. For example, in action-cost minimization settings $g(\pi)$ is the sum of action costs in π and h is the minimal cost of actions that need to inserted. In this mode, for a new plan *best* procedure $Prune$ removes suboptimal partial plans from q, $Prune(q, best) = \{\pi \in q, \sigma(best) > \sigma(\pi)\}$.

- *Satisfaction.* In this mode we are interested only in finding the first plan. We define $\sigma(\pi) = 0$, if π is consistent, $\sigma(\pi) = 1$ otherwise and use the same settings as the optimization mode, which causes immediate pruning of all partial plans upon finding the first plan.

The algorithm is correct by Lemma 4.2.11, any produced plan must have been reached by an application of a sequence of refinements on the initial partial plan, and if it terminates, there are no flaws. However, the algorithm may not terminate since the plan space Π is, in general, infinite. Under the assumptions in Theorem 4.2.12 or Theorem 4.2.13 the algorithm is complete, if we also assume that all the flaws of competing resource timelines are chosen the earliest, and all threats and unbound timelines are chosen the latest. The algorithm is optimal under the same assumptions, and provided with an evaluation function $\sigma(\pi) = g(\pi) + h(\pi)$, if h is admissible. In general, the algorithm is incomplete, as discussed in previous section.

The performance of the planner is critically tied to the realization of procedures $ChoosePlan$ and $ChooseFlaw$. In plan space planning they are commonly known as *flaw selection strategies* and *plan selection strategies*. FAPE is designed to modularly accommodate any number of strategies and allows to create a sequences of strategies $(s_1, .., s_n)$ for both flaw selection and plan selection, such that s_{i+1} works as tie-breaker for s_i.

The flaw selection strategies have been deeply studied in the context of plan space planning and multitude of strategies have been proposed, such as DUnf and DSep (Peot and Smith, 1993), LCFR (Pollack et al., 1997), ZLIFO (Schubert and Gerevini, 1996) and abstract hierarchies in $I_X T_E T$ (Laborie and Ghallab, 1995a). FAPE currently provides the following flaw selection strategies:

- *Least-committing first.* This is a strategy where we choose the flaw with the fewest resolvers. The strategy is identical to the min-domain heuristic in CSP.

- *Abstract Hierarchies.* The concept of the strategy is to assign a *criticality* attribute to each flaw in the plan and then keep choosing the flaws with the highest criticality, where a resolver for a flaw cannot introduce a new flaw with higher criticality. An abstract hierarchy determines the assignments of criticality to the flaws. We construct the hierarchies as described in (Garcia and Laborie, 1996).

The plan selection strategy determines which nodes are expanded first, as such, it merges together various algorithmic approaches to search and heuristics. FAPE provides the following plan selection strategies:

- *Depth-First Search.* Choosing a partial plan π with the maximal value of $depth(\pi)$, where $depth : \Pi \to \mathbb{N}$ counts the number of applied refinements to the partial plan π.

- *Breadth-First Search.* Choosing a partial plan π with the minimal value of $depth(\pi)$.

- *Metric Heuristics.* This is a family of selection techniques that defines $\sigma : \Pi \to \mathbb{R}$ as a function of various attributes of π and chooses the partial plan with the minimal value of $\sigma(\pi)$. FAPE provides the following the variations:

  - *Least Flaw-Action Ratio.* Selects the plans with the smallest ratio $\sigma(\pi) = |flaws(\pi)| \ / \ |actions(\pi)|$.
  - *Linear Combination.* Defines σ as an linear combination of attributes of the partial plan. For example, we define

$$\sigma(\pi) = \\ 10 * |Actions(\pi)| + 3 * |TimelineFlaws(\pi)| + 3 * |OpenLeaves(\pi)|.$$

  The metric strategies can be quite a powerful guidance of the search, we can see them as inadmissible heuristics.

- *Landmark-cut.* This strategy uses an adaptation of an admissible state space heuristic LM-Cut (Helmert and Domshlak, 2011). For a partial plan π, heuristic $h^{lmc}(\pi)$ provides the minimal sum of costs of actions that need to be applied before the partial plan becomes consistent. We formulate $\sigma(\pi) = g(\pi) + h^{lmc}(\pi)$, where $g(\pi)$ is the sum of costs of actions in π (if the costs of actions are not provided, we assume uniform costs of 1). Since LM-Cut is a state-space heuristic, it estimates the distance between two states s_i and s_g. To keep the adaptation of LM-Cut admissible, for a partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ we generate the sets of atoms for s_i and s_g (with respect to Definition 1.0.1) in the following way:

  - $G \subseteq \Phi$, such that:
    - $\forall \phi \in G : \phi$ is an unsupported ground logical timeline with at least one transition,
    - $\forall \phi_i, \phi_j \in G, i \neq j : \phi_i$ does not unify with ϕ_j, and
    - $\forall \phi_i \in G, \phi_j \in \Phi, i \neq j, \phi_i$ unifies with $\phi_j : \phi_i$ occurs necessarily after ϕ_j with regard to N.

    In other words, we take all the latest distinct unsupported ground timelines. State s_g is then constructed as a set of the values of the earliest statements in G.
  - I is a set of ground logical timelines such that:
    - $\forall \phi_i \in I, \exists \phi_j \in \Phi : \phi_i$ is an instance of ϕ_j,

* $I \cap G = \emptyset$.

In other words, I is the set of all instances of timelines in Φ, excluding the ones in G. We generate the state s_i as a set of the values of the latest elements in I.

The construction the sets s_i and s_g does not invalidate the admissibility (with regard to the sum of action costs). The idea of a proof builds on the fact that h^{lmc} operates on a delete-relaxed problem. Therefore, having more atoms achieved in the initial state (because of generating all ground instances of the supported timelines) does not increase the cost, and for the goal state (obtained as initial values of ground unsupported timelines) having less atoms to achieve cannot increase the cost. We can imagine that the adaptation makes a single temporal "slice" of the end of the planning horizon to produce the goal atoms s_g, and then it makes a set of temporal slices for everything that happens before the end slice, producing the initial state s_i as an union of all final values of timelines. We can imagine an example with the following timelines and precedence constraints:

```
A = {x.loc == e :-> f}, where dom(x) = {r1,r2}
B = {r2.loc := a}
C = {r2.loc == c :-> d, r2.loc == d}
B < A, B < C, A < C
```

Then $s_i = \{r1.loc = f, r2.loc = f, r2.loc = a\}$ and $s_g = \{r2.loc = c\}$. In other words, the timeline `A` forms two initial atoms, `B` forms the third initial atom and `C` forms a single goal atom. Therefore we know, that we need to find a path from action statements starting at either $r2.loc = f$ or $r2.loc = a$ and ending at $r2.loc = c$. LM-Cut gives a lower bound on the cost of such path, corresponding to an approximation of the minimal sum of costs of actions producing the path. We can assume such path was found, `A` merged with `C` and we get the following timelines

```
A = {x.loc == e :-> f, r2.loc == f :-> c,
     r2.loc == c :-> d, r2.loc == d}, where dom(x) = {r2}
B = {r2.loc := a}
B < A
```

then $s_i = \{r2.loc = a\}$ and $s_g = \{r2.loc = e\}$. Note that the adaptation of LM-Cut is not monotonic, since merging the timelines can actually increase the estimated cost.

The sets of flaw and plan selection strategies currently provided by FAPE do not represent a complete overview of all available techniques. We have chosen and implemented a selection of techniques that showed a good performance in other systems such as VHPOP (Younes and Simmons, 2003), $I_X T_E T$ (Laborie and Ghallab, 1995a) and theoretical studies such as (Schattenberg, 2009).

4.3.1 Resource Reasoning

The resource reasoning for reservoirs is implemented as an evaluation of the balance constraints (Section 3.2.1), where we always consider only a single production (consumption) event to be a resolver for an overconsumption (overproduction). This is usually a good behavior, since we look for resource conflicts after every refinement and an overconsumption or overproduction is then detected early. Faced with a situations when a single refinement causes such a large overproduction or overconsumption that cannot be balanced by a single action insertion or decomposition, we consider each action insertion or decomposition, that contributes to reducing the overconsumption or overproduction, to be a *partial resolver*. Partial resolvers are treated the same way as resolvers, but they may not resolve a flaw completely. They contribute to making the flaw less difficult to be resolved by resolvers in the future. We can imagine an example when an action `BuildWall` enters the plan and adds an event `bricks :consume 1000`. Then to make the reservoir *bricks* consistent, we need to apply several times a partial resolver `CreateBricks(){ bricks : produce 50}`. Finding an action that contains a resource statement balancing a conflict is a simple problem, since we do not consider any other constraints imposed by the action insertion. Therefore, a lookup table $\{resource\ statement \rightarrow action\}$ serves the purpose.

Given the observation from Filuta (Dvořák, 2009), the Minimal Critical Sets (Section 3.2.2), adapted for reservoirs, tend to quickly start growing exponentially in the number of events (making $Flaws$ procedure to run in exponential time). Therefore, we leave them solely for the discrete resources.

The conflicts on the discrete resources are reasoned about through Minimal Critical Sets, using the Algorithm 10, where each MCS represents a flaw in a plan. In contrast with reservoirs, the resolvers do not contain action and decomposition resolvers, since the overconsumption of a discrete resource cannot be resolved by a new resource event.

The reasoning for producible and consumable resource is trivial, hence we do not go into detail.

4.3.2 Problem Extension and Plan Repair

Planning problems are generally hard and finding a plan represents an investment of a noticeable computational effort. If the planning problem changes even slightly, we would have to invest into finding a different plan again. Such repetitive planning can become problematic in dynamic environments that quickly change and the plan becomes already outdated at the time it is discovered. One of the motivations behind FAPE is to provide an alternative to replanning when the planning problem is modified by new events or the planned actions have not executed properly. We shall now describe these alternatives in detail.

One of the advantages of the chosen representation is the flexibility of the resulting plan. It does not instantiate the objects variables and time points unless there is no other option. When a plan is found for a problem and a new information that modifies the problem appears, we may choose to only update the original problem and try to repair the plan previously discovered. Further, it is not even necessary to start from a consistent plan, but any partial plan can be used to form an initial plan of the updated problem, e.g. when new information

arrives while the planning process is running, and no consistent plan has been found yet, the work invested into building up a partial plan can still worth reusing. We shall now specify how the new information can be integrated.

For a planning problem $P = (\pi_0, O, M, A)$ and any intermediate partial plan $\pi = (N, \Phi, \mathcal{N}, C)$ produced by Algorithm 11 for problem P, we formulate following operations:

- *Object and Type Addition.* We may add any number of new object and types extending O into O', all domains of object variables of the corresponding types become extended by the new objects.

- *Action and Method Addition.* New methods and actions can be added, extending M and A into M' and A'. Using the ANML construction such as *Seed* and $DeusExMachina$ (see Section 4.1), we can also enforce an introduction of new actions into the plan and new statements into the timelines.

- *Removal of an Action.* A removal of an action α from the plan π is performed as follows:

  - We construct $\mathcal{N}' = \mathcal{N} \setminus \{(\alpha, D_\alpha)\}$. In other words, we remove the node corresponding to α from the task network. Note that such removal may cause the task network to loose its legality (Definition 4.2.7).
  - Having $\alpha = (name, P, S, t_s, t_e, C)$, for each statement from $s \in S$ contained in some timeline $\phi \in \Phi$, if s is a condition or a persistent condition (with regard to Definition 4.2.3), we remove it from ϕ. Otherwise we split the timeline ϕ into ϕ_b and ϕ_a, where ϕ_b contains statements strictly before s and ϕ_a contains statements strictly after s with regard to N, and construct $\Phi' = (\Phi \setminus \{\phi\}) \cup \{\phi_b, \phi_a\}$. By Definition 4.2.4, such split is deterministic, since all statements in Φ must have been totally ordered with regard to s.

  The semantics of action removal is dependent on the planning domain. We can imagine a domain where the actions in the decomposition D_α of α represent the necessary side effects and even when α fails and is removed, the side effects remain. In other domains, only leaf node actions may fail. We discuss various semantics of the action removal in Section 4.4.

Any combination of the operations above determines a new partial plan $\pi' = (N, \Phi', \mathcal{N}', C)$ and a new planning problem $P' = (\pi', O', M', A')$, whose solving is expected to be substantially less difficult than solving the initial problem P from scratch with an updated initialization. However, we do not remove any object and temporal constraints as a result of the applying these operations. Therefore, repairing a plan may not lead to a solution, while planning from scratch does. We denote a set of such update operations as U and define a *problem transition function* as follows.

Definition 4.3.1. *For a problem $P = (\pi_0, O, M, A)$, its partial plan π and a set of update operations U we define a problem transition function:*

$$\tau(P, U, \pi) = (\pi', O', M', A')$$

where (π', O', M', A') *is constructed by performing all operations in* U *on* P *and* π*.*

The typical use of the operations above occurs in a situation, when some action in a (consistent) plan π failed to execute in the real-world. In such case, we first remove the action from π and apply any set of collateral effects of the failure as new statements to π. For example, an action *Pick* fails to move an object from a table to a gripper of a robot, in such scenario we remove the action *Pick* from the plan. However, while picking the object, it has fallen on the ground, and such event is then passed as a new statement. An example for the object addition can be an exploration problem, where a robot is traveling through unknown territory, where new objects may appear. When a new object appears, we may stop the planning process, choose a partial plan π from the set q in Algorithm 11, or a consistent plan if one was found, update π, P and start planning again.

Repairing a plan is expected to be less difficult than replanning and especially real-time planning applications can benefit from a better responsiveness of a planning system.

4.3.3 Implementation Details

Most of the techniques we use in FAPE were presented in previous sections, however, some particularities of the implementation are worth to be mentioned.

Discovery of the action resolvers for the unsupported logical timelines can form a bottle-neck, if implemented as an naive search through all actions. To discover actions that contain statements supporting an unsupported timeline, we use the *lifted domain transition graphs* (LDTG) (Jonsson and Bäckström, 1998) and the *relaxed planning graph* (Blum and Furst, 1997) (RPG), which is a common approach in this context. While LDTG operates on a lifted representation, RPG grounds the representation and as such is not suited for domains with large numbers (dozens of thousands) of objects.

A single lifted domain transition graph for a parametrized state variable (corresponding to a declaration of a finite domain function or variable in ANML) can be seen as graph $G = (V, E)$, where V consists of all possible values of the parametrized state variable. Then E contains an arc (x, y) labeled by an action $\alpha = (name, P, S, t_s, t_e, C)$, if there exists a transition statement $T(f, t'_s, t'_e, x', y') \in S$, such that $x' \cap x \neq \emptyset$ and $y' \cap y \neq \emptyset$. Looking for a candidate actions to support an unsupported timeline ϕ, all the actions that set the value v of the earliest transition (or a single persistent condition) in ϕ, are labels of an arc that leads to v in an LDTG corresponding to f.

During planning, we generate a considerable amount of object variables, forming parameters of the state variables and values in statements, and object constraints upon them. The task of finding an instantiation of the object variables such that all object constraints are satisfied is a well known *constraint satisfaction problem* (Dechter, 2003). We use the standard AC-3 (Mackworth, 1977) algorithm to discover quickly an inconsistent problem (implying a dead-end for the search).

FAPE can use either sparse or minimal simple temporal network as described in Chapter 2. Sparse network is however better suited for scenarios, where we

expect to be repairing a plan often. As such, FAPE on default uses the sparse temporal network, maintained by IBFP (Algorithm 9).

The state variable reformulation technique, introduced in Section 1.5, can be used as a preprocessing step of planning to provide further improvement of performance. Since grounding of the a problem is a part of the reformulation, it not suited for domains with large numbers (dozens of thousands) of objects.

The majority of FAPE is implemented in Java, where some components, such as the ANML parser and the sparse temporal network are implemented in Scala. FAPE runs on the Java Virtual Machine 1.7 and higher.

4.4 Acting

While planning can be seen as a service that for a given problem provides some plan, acting needs to model a non-deterministic, partially observable and dynamic environment, control the execution of the actions provided by planning and formulate the planning problems (Ingrand and Ghallab, 2014). FAPE concentrates the acting capabilities into an *Actor* component. The interactions between planning and acting upon a platform is illustrated in Figure 4.3.

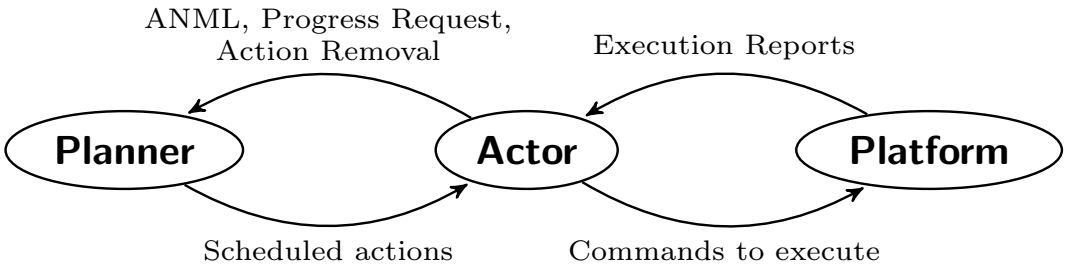


Figure 4.3: The figure shows the integration of acting. Actor component formulates the planning problem through ANML, initiates an instantiation of a plan upon which it receives scheduled actions. The actions are then passed to the platform in form of commands and the platform reports back on their success or failure. A failure of a command leads to a removal of an action from the plan, in which case the planner may need to repair the current plan.

Algorithm 12 represents the high-level view of the implemented Actor component. The input of the algorithm is some initial problem, which can be even empty because the planning challenges have not yet appeared in the environment. The main loop of the algorithm runs until both plan repair and replanning fails, or until it is stopped arbitrarily. In one cycle, we first collect the reports from the platform, those can be new events, successes and failures of commands. In the next step we interpret the reports by translating them into update operations described in Section 4.3.2. At line 9 we construct a new planning problem (see Definition 4.3.1) taking into account all the updates. Procedure $Plan(P, \Delta)$ runs the Algorithm 11 with a timeout Δ. If reparation of the plan fails at line 11, we try to replan at line 13. Having a consistent plan π', we instantiate the plan up to the horizon $\Delta_{horizon}$ and collect all the actions that start before $\Delta_{horizon}$. The

actions, whose start times have fallen into the $\Delta_{horizon}$ for the first time, are then refined into commands and sent to the platform.

The partial instantiation through *Progress* procedure is performed by assigning the lowest possible values to the time points that represent the start times of actions that may possibly start before $\Delta_{horizon}$. The advantage of the partial instantiation is that the plan remains flexible after $\Delta_{horizon}$ and as such is better suited to accommodate potential action failures or new events without a need to replan from scratch. For example a robot carrying an item in each of its two grippers may accidentally drop one of them at some location. Such situation is then introduced to the plan as a failure of the pick action (whose result was that the robot was holding the item), and new statement that sets the current position of the item. Reparation of the plan can consist of moving back and picking up the item again, while the plan for delivering the second item (moving and dropping) remains, but shifts to later time.

The update of $\Delta_{horizon}$ represents synchronization of the real time with the time represented by the temporal network and adding some look-ahead $\Delta_{lookahead}$, representing how far in the future we should be instantiating for. We assume that the input constants of the algorithm satisfy $\Delta_{repair} + \Delta_{replan} + c < \Delta_{lookahead}$, where c is some arbitrary constant (e.g. 100ms) representing an upper bound on the time needed to interpret reports and execute all the parts of the algorithm that are expected to take constant time. We assume there exists a constant $c_{start} \in \mathbb{R}^+$ provided by the platform, representing the moment in real time when the execution of the plan started and corresponding to the time 0 in the temporal network. Additionally, we expect that there exists a variable $c_{current}$ representing the current real time at every moment it is accessed by the algorithm. The updating of the planning horizon is then realized as $Update() = c_{current} - c_{start} + \Delta_{lookahead} + c$. Consequently, we know that the algorithm either fails, or we shall always have the actions that start before $\Delta_{horizon}$ prepared for dispatching.

If an action is reported as a failure, we consider it to be a complete failure and remove it from the plan as described in Section 4.3.2. If the failed action had any impact on the environment, the impact is introduced through arbitrary events, using the `DeusExMachina` construction. Generally, an action that ends too early or too late is considered to be a failure. The failure of an action only causes a removal of the action itself, and its statements from the partial plan. However, the statements could have been providing a support for other statements in the plan, produced by other actions, where some of them might have already been dispatched. The actor component leaves the responsibility for reporting the failures of such actions to the platform. An alternative approach, which we have not adopted, would be to modify the action removal operation to cascade over the statements that became unsupported, removing the dependent actions as well and then sending to the platform some form of "action cancellation" command. The actor component, as it is now, tries to repair after the removal of an action and replans from scratch if reparation fails.

The actor component can only start dispatching actions, if some plan was found. Further, the goals for the planning problems can be introduced from the platform as any goals specified in ANML, be it a level of resource, satisfied condition or decomposed action. For example, an exploration robot can be performing a plan that consists of movement actions and observe actions.

When a new object `book` is identified by the object recognition component of the platform, the existence of the object is passed to the actor as new instance `instance Item book; [start] book.location := floor`, and a new goal can be formed in the platform as `[end] book.location == bookshelf`. The plan incremented with those updates then becomes inconsistent and by repairing the plan the planning algorithm produces a new plan where the transportation of the book occurs.

We have designed several new concepts how to control and delay the decomposition of an action, allowing to fine-tune the refinement process in a way that only the currently relevant actions are decomposed (e.g. initially planning a movement on the level of buildings, but when the start of the movement fits into the $\Delta_{horizon}$, we decompose the movements into navigation between specific locations in space). We discuss the concepts in Future Work.

Algorithm 12 Actor main loop

Input: Formulation of the initial problem $P_0 = (\pi_0, O, M, A)$ in ANML and time constants Δ_{replan}, Δ_{replan} and $\Delta_{lookahead}$ representing how much time we have for repairing, replanning and how far ahead we are looking in the plan for actions to dispatch.

1: $reports \leftarrow \emptyset$
2: $updates \leftarrow \{P_0\}$
3: $\pi \leftarrow \emptyset$
4: $\Delta_{horizon} \leftarrow \Delta_{lookahead}$
5: **while** true **do**
6: $\quad reports \leftarrow$ GetReports
7: $\quad updates \leftarrow updates \cup$ Interpret($reports$)
8: $\quad reports \leftarrow \emptyset$
9: $\quad all \leftarrow all \cup updates$
10: $\quad P' \leftarrow \tau(P, updates, \pi)$
11: $\quad updates \leftarrow \emptyset$
12: $\quad \pi' =$ Plan(P', Δ_{repair})
13: $\quad$ **if** $\pi' = failure$ **then**
14: $\quad\quad \pi' =$ Plan($\tau(P_0, all, \pi_0), \Delta_{replan}$)
15: $\quad\quad$ **if** $\pi' = failure$ **then**
16: $\quad\quad\quad$ **return** $failure$
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: $\quad \Delta_{horizon} \leftarrow$ Update
20: $\quad actions \leftarrow$ Progress($\pi', \Delta_{horizon}$)
21: $\quad$ Dispatch($actions$)
22: $\quad \pi \leftarrow \pi'$
23: **end while**

The actor component does not currently operate with non-determinism (except for possibility of action failures) and uncertainty. Alternative approaches, e.g. based on Markov Decision Processes (Bellman, 1957), are a subject to future development.

5. Experiments

The main focus of FAPE as a planning system is the flexibility of planning. FAPE supports a wide range of features to express various temporal relations and arbitrary events, hierarchical decomposition through methods, explicit reasoning with time and resources, incrementally expendable lifted representation and a reparation of a plan that was arbitrarily broken (e.g. by an action failure). In this chapter we present several setups to evaluate the runtime properties of FAPE and we describe the practical experiments, having FAPE integrated as a planning and acting system in the PR2 robot[1].

5.1 Runtime Properties

FAPE follows the *partial order causal link* (POCL) paradigm (Section 1.3, planning in the space of plans. The extensions towards time, resources and HTN further benefit from planning in plan space, since we can adapt more sophisticated reasoning strategies as opposed to the state space planing (Section 1.1). While the International Planning Competition (Olaya et al., 2011) provides a good selection of planning problems in PDDL (Fox and Long, 2003), it has been for a long time dominated by the state-space forward-search planning systems. We have introduced concepts of some of them in Section 1.5. While POCL planning approaches were intensively studied in decades before the first IPC (1998) and many techniques have been developed, they have not reached the performance to compete with the advancement of the state space planning. Since POCL planning is the base ingredient of FAPE, we do not evaluate it with respect to the performance of IPC planners, but rather showcase the advantages we can gain from the expressiveness in ANML and FAPE.

All the experiments in this section are performed on Intel i5 2.5GHz, 8GB RAM, running on a Java Virtual Machine 1.7. In the rest of this section we present results for scalability of the system with regard to the number of objects, we compare different flaw and plan selection strategies provided by FAPE, we show the advantages of the temporal expressiveness of ANML and how the hierarchical decomposition can benefit the planner.

5.1.1 Scaling in the Number of Objects

One of the advantages of FAPE is planning with the lifted representation. The direct consequence is only a linear dependence on the number of objects in the system as opposed to most of the planning systems in IPC that are grounding the problem on the input, which may grow with polynomial as large as the largest number of arguments of an action or a predicate in the problem.

The real-world planning problems often contain large numbers of objects, while the number of actions in plans remains small. As a practical example we can imagine the airport transportation. There are 41.821 airports in the world[2],

[1] `http://en.wikipedia.org/wiki/Willow_Garage`

[2] `https://www.cia.gov/library/publications/the-world-factbook/fields/2053.html`

and the passengers rarely choose flights with more than two transfers. We model a simplified domain of airport transportation in ANML as follows:

```
type Airport;
type Passenger < object with {
  variable Airport location;};

action fly(Passenger p, Airport a, Airport b){
  [all] p.location == a :-> b;};

instance Passenger p1;
instance Airport prague, barcelona;

[start] p1.location := prague;
[end] p1.location == barcelona;
```

The ground representation of such domain would contain an action for each passenger and combination of airports. Increasing the number of airports to cover all the airports in the world produces more than 1.7 billions of actions for a problem with a single passenger, which a size that does not event fit into a memory of usual personal computer. To evaluate the behavior of FAPE in such scenario, we generate a set of airport transportation problems by fixing the number of passengers to 10, having a single initial location and single destination for each of them, and we vary the number of the airports from 10 to 18.000. Since the problem is trivial from the planning perspective, we do not evaluate it with regard to different configurations of the planner.

Figure 5.1 shows the impact of the number of the airports on total runtime of FAPE and on the time spent planning. The increase of the planning time comes mainly from the need to maintain large domains of object variables. The scaling could be further improved by representing the domains of object variables in other ways than as enumerative sets of values, adopting techniques from constraint satisfaction problems. The increase of the total runtime is mainly caused by the processing of the problem, where parsing the problem contributes the most.

The airport transportation problem could be further extended by adding connectivity between the airports, using the `constant` declaration in ANML. Going further, we can represent planes with some limited capacity as discrete resources, and their flying schedules as arbitrary temporal events. However, such representation scales much worse, since every partial plan would contain all the flying schedules represented as timelines and the temporal network would contain all its time points. Separation of static facts from dynamic facts that are produced by a planner is well known issue in automated planning. If the static facts (state variables) do not change in time, it is common to represent them as a look-up table, outside of the search state. Such concept is also implemented in FAPE. It is more difficult to represent static evolution of a state variable in time, when the search state is heavy, carrying all the temporal developments. Supporting such static timelines is considered to be one of the future goals in FAPE development.

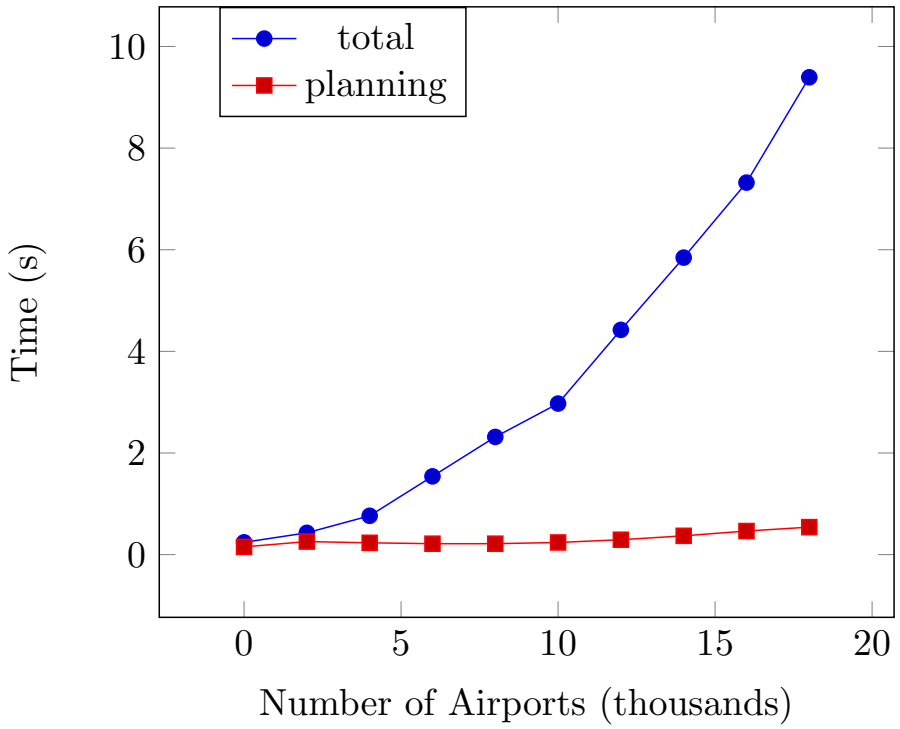


Figure 5.1: The graph shows the dependency of the total runtime (consisting from reading and parsing the file) and the planning time on the number of airports in the *airport transportation* planning problem for 10 passengers. The problem remains the same, while the number of the airports increases. The problem is simple, all flights are direct flights. Planning time itself increases from 100ms up to 500ms.

5.1.2 Performance for Different Domains

There has not yet been created a set of testing data in ANML with focus on performance evaluation. For the purpose of giving a lookahead on the expected performance of FAPE and some of its implemented strategies, we generate four sets of problems, where three of them are translations of PDDL domains *elevators*, *rovers* and *visitall* from IPC (Olaya et al., 2011), and the domain *handover* corresponds to one of the setups of practical experiments with robots.

The elevators planning domain consists of a set of passengers, elevators and floors. Some of the floors are connected and some of the floors can be reached only by some elevators. The elevators have a limited capacity and the goal is to transport all the passengers from their initial locations to their final destinations.

The rovers planning domain is motivated by space applications and models the exploration of a planetary surface by a set of rovers of different capabilities. The domain is quite complex, the goals consist of analyzing soil and rock samples at certain locations and taking images of some locations (taking an image of a location A requires to stay at location B, where A is visible from B). Further, only some robots can performs soil analysis, or rock analysis and only some of them have a camera to take images.

The visitall domain represents an exploration robot, whose goal is to visit a set of locations. The number of locations reaches one hundred and the connectivity

graph can be both sparse or dense.

The handover domain represents a robotic experiment, where two robots need to deliver an item to a its destination by cooperation. The domain consists of robots, items and locations, where some of the locations are connected (robot can move between them), and some locations are close enough, such that two robots can pass an item from one two another (`passable(Location a,Location b)`).

For each of the domains we generate (translate) 20 problem instances. Further, we fix the flaw selection strategy as follows:

- All flaws of competing resource timelines are chosen the first.
- All threats and flaws of unbound timelines are chosen the latest.
- Flaws are chosen by least-commitment first strategy and abstract hierarchical tree are used as a tie-breaker, as described in Section 4.3.

We use five plan selection strategies:

- *lmc* denotes the landmark-cut plan selection strategy.
- *fa* denotes a metric strategy, computed as:

  $$\sigma(\pi) = 10 * |Actions(\pi)| + 3 * |TimelineFlaws(\pi)| + 3 * |OpenLeaves(\pi)|.$$

- *lmc>fa* denotes using landmark-cut as main strategy and fa as a tie-breaker.
- *lmc>lfr* denotes using landmark-cut as main strategy and least flaw-action ratio as a tie-breaker.
- *fa>lfr* denotes using fa as a main strategy and lfr as a tie-breaker.

We generate (translate) 20 problem instances for each of the described domains and run FAPE in satisfaction mode for each problem and each plan selection strategy with a one minute timeout. The results are split into three figures, Figure 5.2 shows how problem instances were solved for each domain and plan selections strategy, Figure 5.3 shows how many search states were expanded for all problems in the given domain that were solved using all the strategies and Figure 5.4 shows total planning time per domain and planner, considering only the problems solved by all strategies.

	fa	lmc	lmc>fa	lmc>lfr	fa>lfr
handover	19	**20**	**20**	**20**	19
rovers	5	5	5	5	5
elevators	12	**19**	17	17	12
visitall	**8**	7	7	7	**8**

Figure 5.2: Table shows the number of solved problems for different plan selection strategies and domains. There were 20 problem instances for each domain.

	fa	lmc	lmc>fa	lmc>lfr	fa>lfr
handover	16628	47458	29477	27406	**16535**
rovers	870	4527	1111	1101	**763**
elevators	13386	6833	**5676**	5720	12601
visitall	1434	895	**722**	**722**	1394

Figure 5.3: Table shows the total number of generated search states for each domain and plan selection strategy. Considering only those problem instances that were solved by all strategies.

	fa	lmc	lmc>fa	lmc>lfr	fa>lfr
handover	16.527	120.634	83.419	72.124	**12.238**
rovers	2.739	26.306	5.295	5.575	**1.599**
elevators	46.925	46.189	**33.853**	34.719	45.416
visitall	1.287	1.55	**0.314**	0.367	0.232

Figure 5.4: Table shows the sum of total planning times needed to find a solution for each domain and plan selection strategy. Considering only those problem instances that were solved by all strategies.

We can notice that the fa strategy by itself leads the search algorithm very quickly to a solution. It expands considerably less states states and aided with lfr as tie-breaker this effect is even stronger. However, it discovers fewer solutions in handover domain and considerably fewer solutions in elevators domains, since it can lead the search algorithm into a branch full of dead-ends. The landmark-cut heuristic (lmc) can estimate the distance from the goal state quite well in the state space. However in plan space we also make large numbers of decisions that do not involve insertions of actions, leading to a creation of large plateaus, where the heuristic does not guide at all. For example a plan with 10 actions in the elevators domain is usually found in search depth around 80, but landmark-cut cannot guide on better granularity than [0,10] (assuming action with unary cost). We can see that those plateaus can be avoided, if lmc uses another strategy for tie-breaking, giving better across all domains, except for elevators, where pure lmc performs better. lmc without a tie-breaker can be seen as a breadth-first search with an occasional descent to a better value of lmc. Elevators domain contains large amount of dead-ends caused by resource inconsistencies (it may take longer to discover that capacity of the elevator was overconsumed), therefore fa guidance, although it is leading to a solution faster, may cause the search algorithm to get lost in an branch that contains only dead-ends.

Development of further heuristics and strategies is one of the goals of further development, especially strategies that take into consideration criticality of resource conflicts and relate the flaws of resource timelines with flaws of the logical timelines. Additionally, we have used a fixed flaw selection strategy but many more were developed, especially domain-specific flaw selection strategies, and it is a common practice in POCL planning, e.g. in VHPOP (Younes and Simmons, 2003), to generate states for multiple flaw selection strategies at once. We plan conduct further experiments for flaw selection as well.

5.1.3 Expressiveness Examples

We shall further showcase some of the advantages of the expressiveness of FAPE and ANML, starting by describing the elevators domain and then going into detail on how can the hierarchies and temporarily expressive statements improve the performance by adding more knowledge of the problem.

We formulate the ANML problem in a similar way as the original PDDL formulation. The domain of the problem is described as follows.

```
type Location; type Floor < Location;
type Passenger < object with {
  variable Location location;};
type Elevator < Location with {
  variable Floor location;
  replenishable integer [0,infinity] occupancy;};

constant boolean connected(Floor a, Floor b);
constant boolean reachable(Elevator e, Floor a);

action board(Passenger p, Elevator e, Floor l) {
  end = start + 1;
  [all] {
    p.location == l :-> e;
    e.location == l;
    e.occupancy :consume 1;};};

action leave(Passenger p, Elevator e, Floor l) {
  end = start + 1;
  [all] {
    p.location == e :-> l;
    e.location == l;
    e.occupancy :produce 1;};};

action move(Elevator e, Floor a, Floor b) {
  end = start + 10;
  [all] {
    reachable(e,a) == true;
    reachable(e,b) == true;
    connected(a,b) == true;
    e.location == a :-> b;};};
```

The domain is straightforward, the passenger can board an elevator or leave it, and the elevator can move between floors. The elevator has some limited capacity that represents the number of passengers that can fit into the elevator. We set up a problem as follows.

```
instance Floor f1,f2; instance Elevator e1; instance Passenger p1;

[start]{
  e1.occupancy := 2; e1.location := f1; p1.location := f1;
```

```
  reachable(e1,f2) := true; reachable(e1,f1) := true;
  connected(f1,f2) := true;};

[end]{
  p1.location == f2;};
```

In other words, we declare the floors, elevators, passengers. Initial locations of passengers and elevators, the initial capacity of elevators, reachability and connectivity of floors and the goal destination for passengers. We shall now show how to alternate the original representation.

Hierarchical Formulation

The HTN planning (Section 1.4) plans by decomposing actions (tasks) through methods and any resulting plan must have necessarily been constructed by a sequence of decompositions. In FAPE this requirement is relaxed, plan can be formed by a combination of decompositions and action insertions. Such concept can become quite powerful, since we do not need to specify every possible corner case through decompositions, but only the main concept of it and leave the corner cases to be covered by action insertions. Taking as an example the elevators domain, we know that if the passenger wants to get somewhere, he needs to enter an elevator, let the elevator move between floors and then leave. Such knowledge can be encoded by the following ANML action.

```
action transport(Elevator e, Passenger p, Location a, Location b){
  :decomposition{
   ordered(
      board(p,e,a),
      leave(p,e,b));};};
```

Assuming that all the passengers need to travel somewhere (otherwise they would not be a part of the problem), we know that each passenger will be transported at least once. We can encode that by adding a transport action for each passenger to the initial plan as follows.

```
action Seed(){
  :decomposition{
    transport(anyElevator, p1, anyFloor, f2);};};
```

Such extension can be done without altering the original problem in anyway, except for removing the goal statements, which are now encoded as a part of the transport actions. The transport action is a form of shortcut in the search space. Such concept is also known as *macro action*, first seen in (Fikes et al., 1972), yet hierarchies allow to structure the problem on multiple levels of abstraction. Figure 5.5 shows the impact of an alternative formulation on the runtime performance of planning.

Temporarily Expressive Formulation

The expressiveness of ANML allows us to go further and formulate the elevators problem more compactly. Instead of using three actions and a reservoir for modeling the capacity of the elevator, we can use only two actions and a discrete resource, using the following formulation.

```
action move(Elevator e, Floor a, Floor b) {
  end = start + 10;
  [all] {
    reachable(e,a) == true;
    reachable(e,b) == true;
    connected(a,b) == true;
    e.location == a :-> b;};};

action useElevator(Passenger p, Elevator e, Floor a, Floor b) {
  [all] {
    reachable(e,a) == true;
    reachable(e,b) == true;
    connected(a,b) == true;
    e.occupancy :use 1;
    p.location == a :-> b;};
  [start,start+1] e.location == a;
  [end-1,end] e.location == b;};
```

We know that entering and leaving the elevator takes 1 time unit, also we know, that a passenger who enters an elevator eventually leaves the elevator at some floor. Therefore, we can capture how the elevator is used in one action `useElevator`, while the action `move` is the same as original. This formulation also removes the need to represent explicitly the location of the passenger in the elevator, since it is recorded only as a resource usage. In Figure 5.5 we can notice, that the new formulation of the problem leads to improved performance. The performance gain is caused simply by reducing the number of actions needed to achieve a plan, and using a discrete resource instead of a reservoir, which reduces the number of generated flaws. A path between floors of a single passenger is constructed from actions `useElevator`, where each of them needs to be supported by an action `move`. A single instance of action `move` can support multiple instances of action `useElevator`. The sharing represents that multiple passengers were in the elevator, and the discrete resource `occupancy(Elevator)` ensures that those passengers could fit into the elevator. The domain is simplified by using constant durations for actions, if the duration of the move action is linearly dependent on the distance between floors, we can imagine situations, when a single `useElevator` gets supported by two or more different actions `move`. The power of the expressive temporal annotations can be manifested in a multitude of common temporal planning problems, and while FAPE cannot currently compare to the efficiency of IPC planning systems on IPC planning problems in PDDL, there are domains, such as elevators, where the additional expressiveness can greatly simplify the problem.

5.2 Practical Experiments

The focus of the practical experiments has been to integrate FAPE into PR2 robot, allowing the robot to be controlled by FAPE in real-time.

We use the acting component introduced in Section 4.4, where the planning procedure is realized by the Algorithm 11, run with a timeout given by the ac-

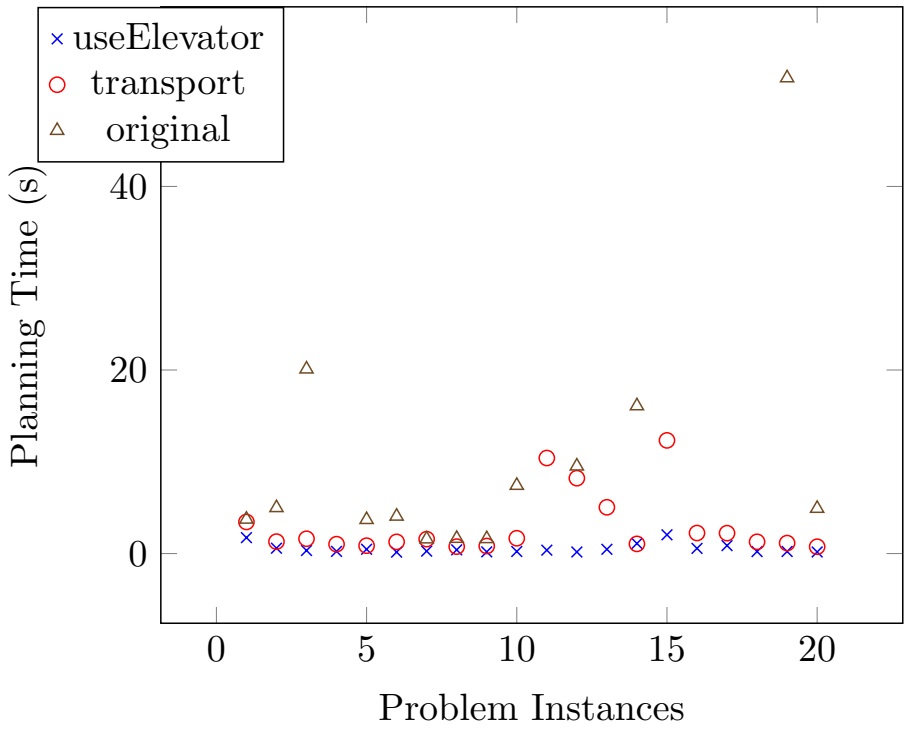


Figure 5.5: The figure shows the planning time (time needed before a first plan is found) for different instances of problems in the elevators domain, using one set of the *original* problems, one set of problems reformulated with action *transport* and another set reformulated using the action *useElevator* instead of *board* and *leave*.

tor. The action instances produced by the planner are refined into collections of closed loop commands that control the motor forces of PR2 robot. The mapping between the action instances and the commands is implemented through hand-written procedure in PRS language (Ingrand et al., 1996). PRS also contains the interpretation for the information perceived in the environment and translates the perceived information to reports on action successes, failures and new arbitrary events. PRS uses the basic motor commands and perceptions of the environment implemented in ROS[3] and GenoM3 (Mallet et al., 2010). Specifically, the navigation and localization is provided by ROS PR2 Navigation stack and the object perception, recognition and manipulation planning is provided by dedicated modules of GenoM, developed in LAAS-CNRS[4]. FAPE has been run as a two threaded application, where one thread was occupied by the acting component that also runs the planning algorithm, and the other thread run a listing loop that was receiving reports through a socket connection to PRS Message Passer.

The environment used for the experiments was an apartment (Figure 5.6) in LAAS with several rooms and various static and movable objects. The setup of the experiments corresponds to tasks a service-robot could perform. For example, the PR2 moves around in an apartment and detects objects which are misplaced (e.g. a video tape in the bedroom, or a book on the dining table) picks them up

[3] `http://wiki.ros.org/Robots/PR2`

[4] Laboratoire d'Analyse et d'Architecture des Systèmes `http://www.laas.fr`

and stores them away in their proper location (respectively by the TV set, and in the bookshelf).

Figure 5.6: PR2 robot in the apartment environment.

The most simple formulation of planning problem consists of a single robot with two grippers, multiple locations and several objects that need to be transported. Such *apartment problem* can be formulated in ANML as follows.

```
type Location;

type Gripper < Location with {
  variable boolean empty;};

type Robot < object with {
  variable Location location;
```

```
  variable Gripper left;
  variable Gripper right;};

type Item < object with {
  variable Location location;};

action move(Robot r, Location a, Location b){
    [all] r.mLocation == a :-> b;};

action pickLeft(Robot r, Gripper g, Item i, Location l){
  [all]{
    r.left == g;
    r.location == l;
    g.empty == true :-> false;
    i.location == l :-> g;};};

action dropLeft(Robot r, Gripper g, Item i, Location l){
  [all]{
    r.left == g;
    r.location == l;
    g.empty == false :-> true;
    i.location == g :-> l;};};
```

Further extensions of the domain included two robots, swapping items between different grippers, plugging the robots to a power outlet and various hierarchies, e.g. having a pick action decompose to either picking by left or right gripper.

We have observed that FAPE, provided with an ANML problem, using one second timeout for planning ($\Delta_{replan} = \Delta_{repair} = 1s$) and 10 seconds planning horizon ($\Delta_{lookahead} = 10s$), was able to control the PR2 robot according to the plan, move it around the apartment, perform an action and recover from an action failure. The practical experiments demonstrated the functionality of FAPE as a control system. The experiments themselves did not really stress the planning performance of FAPE, since there was usually a plenty of time to plan. In particular, the execution of actions took seconds (e.g. a robot moving), while the planning took less than a second, having plans of less than ten actions.

Since the planning with the robot did not tap into the planners performance capabilities, to evaluate the performance benefits of the plan reparation, we have written a simple simulator of PR2 in the PRS, having a 50% probability of failure for every action instance produced by the planner. In each case of failure we evaluate the performance of planning from scratch and reparation of the plan with regard to the number of search nodes (partial plans) that needed to be generated.

Figure 5.7 shows that in 82.2% of the situations the reparation generates less than 15 nodes, being almost instantaneous. Further, only in 4.8% of cases the reparation is more demanding than planning from scratch and the plan reparation fails in 7.3% cases. Planning from scratch generated 319 search nodes on average.

	Number of Instances
$n_{repair} \leq 15$	102 (82.2%)
$15 < n_{repair} < n_{replan}$	7 (5.7%)
$n_{repair} \geq n_{replan}$	6 (4.8%)
repair failed	9 (7.3%)

Figure 5.7: Comparison of planning from scratch and repairing the plan in case of action failures, generated by PRS simulation. n_{repair} represents the number of generated search states (partial plans) when repairing the plan, n_{replan} represents the same when planning from scratch.

5.3 Summary

FAPE is an expressive planning system that can be embedded as a planning and acting system. The expressiveness of FAPE, aided by the language ANML, is quite wide and comparable only to a few planning systems, such as $I_X T_E T$ (Laborie and Ghallab, 1995a) and *EUROPA* (Frank and Jónsson, 2003), which also integrate explicit temporal reasoning, resources, and some form of *macro actions* that can be seen as a variation of HTN methods.

Most of the planning systems use some simplifying assumptions that help to face the complexity of planning problems. The simplification of time representation into instantaneous events in classical planning is such an example. The representation of resources is also often simplified, e.g. using a propositional encoding as we can find in some problems of the IPC (Olaya et al., 2011). Some systems, such as CPT (Vidal and Geffner, 2006), assume that an action cannot occur more than once at any plan. Such simplification especially favors constraint satisfaction representations of a planning problem, putting an artificial limit on the size of the plan. FAPE is avoiding most of such simplifications, and the strongest assumptions it uses are considering only resources with piece-wise constant development and totally ordering the transitions in timelines in Definition 4.2.4. The impact of the ordering assumption does not effect the expressiveness, but represents a concept how to maintain the causality between events in time. However, it is more committing than maintaining causal links explicitly, as seen in plan space planning (Section 1.3, because the ordering introduces additional temporal constraints into a partial plan.

Given the level of expressiveness in FAPE, it is not directly comparable to the planning systems appearing in IPC. Using the planning problems from IPC, FAPE cannot reach the performance of those planners. On the other hand, problems with heavy resource reasoning and required temporal concurrency would be difficult to solve by the IPC planning systems. We perform several sets of experiments to evaluate the runtime properties and planning capabilities of FAPE. The scalability in number of objects contributes to the long term movement of domain independent planning towards solving real-world problems. Merging of plan space planning and HTN planning then retains the benefits of both. FAPE can be used as a pure HTN planner and also as a plan space planner. We experimentally demonstrate that even staying between the two, allowing both action insertions and decompositions, can improve the performance. Finally, we demonstrate that

improvement of performance can be gained from the temporal expressiveness in ANML.

The goal of the practical experiments has been to verify that FAPE is capable of acting as a control system. For that purpose, we have performed experiments on a real robot, controlled FAPE, where the robot was able to act according to plan, recover from a failure by repairing the plan and introduce unexpected events into a plan. Since the real-world experiments did not really tap the planning performance of FAPE, we have also performed a set of close-to-real-world experiments that showed the benefit of plan reparation.

Conclusion

In this thesis we have thoroughly investigated different planning approaches, heuristics and representations for classical planning. Based on the findings, we have proposed a novel method *h2-greedy* for constructing the finite-domain state-variable representation of planning problems. Using the planning systems and problems from IPC (Olaya et al., 2011), we have performed an extensive number of experiments showing that *h2-greedy* produces representations competitive with currently dominant translation techniques and provides significant improvement of performance for some planning systems and planning domains. The improvement becomes apparent for planning systems that internally use the state-variable representation. In some domains *h2-greedy* even reduces the size of the problem leading to improved performance across all planning systems. The method is fully automated and can be used as a stand-alone preprocessing tool for any planning system.

In second chapter we have investigated the extension of planning with an explicit time representation, suited for integration of planning and resource reasoning. We have further focused on the simple temporal networks and different algorithms for maintaining the consistency of the network. Using a set of generated temporal reasoning scenarios, we have identified the transition of performance between sparse and minimal representations. The results can be further used for development of hybrid strategies that switch between the two representations, leading to an overall improved performance of temporal reasoning.

We have focused on the roles resources play in planning and scheduling in the third chapter. We have identified different behaviors of resources and proposed their categorization. After formally defining resources as *resource temporal networks*, a representation that unites the view of the resource across planning and scheduling, we have investigated different techniques for reasoning with resources. We have identified the differences between the roles resources play in planning and scheduling and presented two reasoning techniques suited for maintaining consistency of *discrete resources* and *reservoirs* in planning.

The previous chapters built up the techniques for presenting a new planning system FAPE in the fourth chapter. FAPE is planning system that integrates plan-space planning, explicit time representation, resource reasoning and hierarchical decompositions of actions. It is the first system that implements the majority of features in the language ANML, used as an input of the system. FAPE can be used solely for offline planning, but it also integrates acting, aided by efficient plan reparation and maintenance of plan flexibility, allowing it to operate as an embedded real-time planning and acting system. FAPE is designed to be easily extendable with alternative time and resource reasoning techniques, and new plan and flaw selection strategies. As such, we can see FAPE as a planning framework, already implementing multitude of techniques and providing a large set of configuration options. Evaluating runtime properties of FAPE, we showed that we can gain significant performance improvement from hierarchical decompositions and the temporal expressiveness of ANML compared to a common formulation of planning problems in PDDL. In the real-world experiments we have demonstrated that FAPE can operate as a planning and acting system

and the close-to-real-world experiments confirmed the benefits of plan reparation and plan flexibility with regard to the performance of the system. Having FAPE embedded as a planning and acting system may significantly simplify the development of new behaviors – instead of scripting or hard-coding we may declare an ANML problem and let the actions to be planned.

Future Work

FAPE is a system that spans over a multitude of topics, where all of them deserve attention. Starting with reformulation technique introduced in Section 1.5 (which can be also considered as stand-alone tool), there is still a space for improvement by adapting other techniques for mutex generation and clique covering. In (Dvořák et al., 2013) we have also observed that overcovering may lead to better performance, forming another possible direction to investigate.

The performance of the temporal reasoning in FAPE is currently quite satisfying, however the flexibility of the simple temporal network is limited, when we are faced with uncontrollable temporal relations, such as actions, whose duration varies and is dependent on unpredictable factors. The duration of such actions can be modeled by the upper bound on their duration, expecting that if they end sooner, the plan will be still valid. Such assumption is not always correct, e.g. finishing cooking the diner sooner will lead to eating a cold diner. Such situations can be modeled by *simple temporal problems with uncertainty* (Vidal and Ghallab, 1996), whose integration into FAPE is being currently pursued.

The resource reasoning in FAPE is currently limited to piece-wise constant resources, which is a simplification of the actual resource consumption, e.g. the fuel is consumed continually, but since the amount of the fuel between the start and the end of the consumption does not play any role, it is safe to simplify. This might not be a case in other situations, e.g. a battery with a set of concurrent linear consumptions and productions that must always provide enough energy. We are aware of a single planning system COLIN (Coles et al., 2012) that supports such resource and its addition to FAPE is a subject to future development.

The quality of flaw and plan selection strategies has a significant impact on FAPE performance. Currently, the flaws represent a unified view for unsupported logical and inconsistent resource timelines. However the common strategies for flaw selection in plan space planning do not consider resources at all. The investigation of flaw selection strategies, especially focused on their interactions, is strong direction for the future. Further, we plan to investigate performance of other adaptations of state-space heuristics for plan selection, where the *additive heuristic* (Blum and Furst, 1997) as implemented in VHPOP (Simmons and Younes, 2011) is a first candidate. Another direction are the heuristics that predict the usage of resources, and finally, we would like to extend the perception of the plan quality towards makespan (total plan execution time) and other temporal metrics.

We plan to extend the acting component with other refinement concepts, such policies based on Markov decision processes (Morisset and Ghallab, 2008) and Dynamic Bayes Networks (Infantes et al., 2011), and further investigate different approaches for interleaving acting and planning.

Another interesting direction of investigation and development are the control mechanisms for hierarchical decompositions. We have designed but not yet experimentally tested new control mechanics for decomposition that bring the domain designer more power to fine tune the search and also provide more support for embedded planning. All of the extensions are part of method definition, we call those extensions *hard*, *soft* and *weak*.

- The hard extension is an additional condition (a temporal statement) that tells the planner if the method needs to be decomposed (if the condition does not hold then we do not decompose the method and it does not invalidate consistency). The extension allows a multitude of control use-cases to be introduced, e.g. we may start decomposing certain methods only once we get close to their execution — this is the case for the navigation action that can be abstracted as a motion from a to b, until we approach the time of the action and need refine it into a sequence of path following actions that would be otherwise unnecessary to keep in the plan in advance.

- The soft extension allows us to define priorities of decompositions — we simply assign a priority to every method then we try expand those with the highest priority first, we can see this extension as an explicit heuristic entered by the domain designed or the real-time environment.

- The weak extension represents a look ahead for a decomposition of a method, its main purpose is to propagate new time bounds and constraints. Having a method with several possible decompositions (we call the regular decompositions hard), we add at most one weak decomposition. The weak decomposition method is then always performed when we add the method to the plan and it is non-colliding with any hard decomposition that is chosen later during the search.

Another mechanic introduced in ANML and implemented in FAPE is the restriction on action insertions called `motivated`. Any action marked with the keyword can be only inserted to the plan by a decomposition. This allows to further fine-tune the behavior of decompositions, e.g. some actions are only meant to be used together with other actions such as action *Charge* and *Release* for an air-powered tool, where an usage of the *Charge* action out of context would cause damage to the tool.

FAPE is being continually developed and we plan to release it publicly in near future.

Bibliography

Ruth Aylett, Gary J. Petley, Paul W. H. Chung, B. Chen, and David William Edwards. Ai planning: solutions for real world problems. *Knowledge-Based Systems*, 13(2-3):61–69, 2000.

Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–656, 1995.

Philippe Baptiste and Claude Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 600–606, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8, 978-1-558-60363-9.

Peter Van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *Proceedings Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 585–590, 1999.

Richard Bellman. A Markovian Decision Process. *Indiana University Mathematics Journal*, 6:679–684, 1957.

Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of 25th Annual ACM Symposium on Theory of Computing, ACM*, pages 11–20, 1993.

Susanne Biundo and Bernd Schattenberg. From abstract crisis to concrete relief – a preliminary report on combining state abstraction and htn planning. In *Proceedings of the European Conference on Planning*, pages 157–168. Springer Verlag, 2001.

Avrim Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. volume 90, pages 281–300, 1997.

Mark S. Boddy and Daniel P. Johnson. A new method for the global solution of large systems of continuous constraints. In Christian Bliek, Christophe Jermann, and Arnold Neumaier, editors, *COCOS*, volume 2861 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2002. ISBN 3-540-20463-6.

Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The Maximum Clique Problem. *Handbook of Combinatorial Optimization*, 4: 1–74, 1999. doi: 10.1.1.56.6221.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719. MIT Press, 1997.

Ravi Boppana and Magnús M. Halldórsson. Approximating Maximum Independent Sets by Excluding Subgraphs. *BIT*, 32(2):180–196, 1992. ISSN 0006-3835. doi: 10.1007/BF01994876.

Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69:165–204, 1994.

A. Cesta and A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. In *Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*, TIME '96, pages 45–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7528-4.

Yixin Chen, Benjamin W. Wah, and Chih wei Hsu. Temporal planning using subgoal partitioning and resolution in sgplan. *Journal of Artificial Intelligence Research*, 26:369, 2006.

A Coles, A Coles, Maria Fox, and Derek Long. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research*, 44:1–96, 2012.

Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Colin: Planning with continuous linear numeric change. *CoRR*, abs/1401.5857, 2014.

Gregg Collins and Louise Pryor. Planning under uncertainty: Some key issues. In *IJCAI*, pages 1567–1575. Morgan Kaufmann, 1995.

Ken Currie and Austin Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, November 1991. ISSN 0004-3702. doi: 10.1016/0004-3702(91)90024-E.

Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1558608907.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

C. Domshlak, M. Helmert, E. Karpas, E. Keyder, S. Richter, J. Seipp, and M. Westphal. BJOLP: The Big Joint Optimal Landmarks Planner. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 91–95, 2011a.

C. Domshlak, M. Helmert, E. Karpas, and S. Markovitch. The SelMax Planner: Online Learning for Speeding up Optimal Planning. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 108–112, 2011b.

Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Multivalued Action Languages with Constraints in CLP(FD). In *Proceedings of the 23rd International Conference on Logic Programming*, ICLP'07, pages 255–270. Springer-Verlag, 2007.

Filip Dvořák. AI Planning with Time and Resource Constraints. *Diploma Thesis*, pages 1–84, August 2009.

Filip Dvořák, Daniel Toropila, and Roman Barták. Towards AI Planning Efficiency: Finite-Domain State Variable Reformulation. In *Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation, SARA 2013*. AAAI, 2013.

Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th Europen Conference on Planning (ECP-01)*, pages 13–24, 2001.

Stefan Edelkamp and Peter Kissmann. GAMER: Bridging Planning and General Game Playing with Symbolic Search. In *Proceedings of IPC-6*, 2008.

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, Decidability and Undecidability Results for Domain-Independent Planning. *Artificial Intelligence*, 76(1-2):75–88, 1995a.

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artif. Intell.*, 76 (1-2):75–88, 1995b.

Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *ICAPS*. AAAI, 2009. ISBN 978-1-57735-406-2.

C. Fawcett, M. Helmert, H. Hoos, E. Karpas, G. Röger, and J. Seipp. FD-Autotune: Automated Configuration of Fast Downward. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 31–37, 2011.

Richard E Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1972.

Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

Lucian Finta, Zhen Liu, and Centre Sophia Antipolis. Single machine scheduling subject to precedence delays. *Discrete Applied Mathematics*, 70:247–266, 1995.

Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20: 61–124, 2003.

Jeremy Frank. Bounding the resource availability of partially ordered events with constant resource impact. In *Proceedings of the 10th International Conference on Principles and Practices of Constraint Programming*, 2004.

Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Journal of Constraints, Special Issue on Constraints and Planning*, 8:339–364, 2003.

Frédérick Garcia and Philippe Laborie. New directions in ai planning. chapter Hierarchisation of the Search Space in Temporal Planning, pages 217–231. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1996. ISBN 90-5199-237-8.

Alfonso Gerevini and Lenhart K. Schubert. Inferring State Constraints for Domain-Independent Planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98*, pages 905–912. AAAI Press / The MIT Press, 1998.

Malik Ghallab, Dana S Nau, and Paolo Traverso. *Automated Planning: Theory and Practice.* Morgann Kaufmann, October 2004.

Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239. Morgan Kaufmann, 1969.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, (37): 28–29, December 1972. ISSN 0163-5719. doi: 10.1145/1056777.1056779.

William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, pages 607–613. Morgan Kaufmann, 1995.

Patrik Haslum and Hector Geffner. Admissible Heuristics for Optimal Planning. In *Proceedings of 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 140–149, 2000.

M. Helmert and C. Domshlak. LM-Cut: Optimal Planning with the Landmark-Cut Heuristic. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 103–105, 2011.

M. Helmert, G. Röger, J. Seipp, E. Karpas, J. Hoffmann, E. Keyder, R. Nissim, S. Richter, and M. Westphal. Fast Downward Stone Soup. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 38–45, 2011.

Malte Helmert. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*. AAAI, 2009. ISBN 978-1-57735-406-2.

Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, pages 176–183, 2007.

Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22:57–62, 2001.

Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A Novel Transition Based Encoding Scheme for Planning as Satisfiability. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010*. AAAI Press, 2010.

Yong K. Hwang and Narendra Ahuja. Gross motion planning, a survey. *ACM Computing Surveys*, 24(3):219–291, September 1992. ISSN 0360-0300. doi: 10.1145/136035.136037.

Guillaume Infantes, Malik Ghallab, and Félix Ingrand. Learning the behavior model of a robot. *Autonomous Robots*, 30(2):157–177, February 2011. ISSN 0929-5593. doi: 10.1007/s10514-010-9212-1.

F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 43–49, Minneapolis, 1996.

Félix Ingrand and Malik Ghallab. Robotics and artificial intelligence: A perspective on deliberation functions. *AI Communincation*, 27(1):63–80, 2014.

Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, 1998.

R. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier, editor, *IJCAI*, pages 1728–1733, 2009.

Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, pages 174–181. AAAI, 2008. ISBN 978-1-57735-386-7.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

Philippe Laborie. Resource temporal networks: Definition and complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 948–953, San Francisco, CA, USA, 2003a. Morgan Kaufmann Publishers Inc.

Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003b.

Philippe Laborie and Malik. Ghallab. Ixtet: an integrated approach for plan generation and scheduling. In *4th IEEE-INRIA Symposium on Emerging Technologies and Factory Automation*, SC '12, pages 485–495, 1995a.

Philippe Laborie and Malik Ghallab. Planning with sharable resource constraints. In *International Joint Conference on Artificial intelligence (IJCAI)*, pages 1643–1649, 1995b.

Derek Long, Maria Fox, Laura Sebastia, and Alex Coddington. An examination of resources in planning. In *Proceedings of 19th UK Planning and Scheduling Workshop, Milton Keynes*, 2000a.

Derek Long, Henry A. Kautz, Bart Selman, Blai Bonet, Hector Geffner, Jana Koehler, Michael Brenner, Jörg Hoffmann, Frank Rittinger, Corin R. Anderson, Daniel S. Weld, David E. Smith, and Maria Fox. The AIPS-98 Planning Competition. *AI Magazine*, 21(2):13–33, 2000b.

Pierre Lopez. *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources 1 microfiche - papier*. PhD thesis, Toulouse 3, 1991.

Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2010.

Frederic Maris and Pierre Régnier. Tlp-gp: New results on temporally-expressive planning benchmarks. In *ICTAI (1)*, pages 507–514. IEEE Computer Society, 2008.

David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In *AAAI*, pages 634–639. AAAI Press / The MIT Press, 1991. ISBN 0-262-51059-6.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

Benoit Morisset and Malik Ghallab. Learning how to combine sensory-motor functions into a robust behavior. *Artificial Intelligence*, 172(4-5):392–412, March 2008. ISSN 0004-3702. doi: 10.1016/j.artint.2007.07.003.

Nicola Muscettola. Computing the envelope for stepwise-constant resource allocations. In *Proceedings of the 9th International Conference on the Principles and Practices of Constraint Programming*, pages 139–154. Springer, 2002.

Dana Nau, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *IJCAI*, pages 459–466. Morgan Kaufmann, 2001. ISBN 1-55860-777-3.

R. Nissim, J. Hoffmann, and M. Helmert. The Merge-and-Shrink Planner: Bisimulation-based Abstraction for Optimal Planning. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pages 106–107, 2011.

W.P.M. Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1994.

A. Olaya, C. Lopez, and S. Jimenez. International Planning Competition. *ICAPS 2011*, 2011. URL `http://ipc.icaps-conference.org/`.

J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl. pages 103–114. Morgan Kaufmann, 1992.

Mark A. Peot and David E. Smith. Threat-removal strategies for partial-order planning. In *AAAI*, pages 492–499. AAAI Press / The MIT Press, 1993. ISBN 0-262-51071-5.

David Pizzi, Jean-Luc Lugrin, Alex Whittaker, and Marc Cavazza. Automatic generation of game level solutions as storyboards. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(3):149–161, 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2070066.

Leon R. Planken. Incrementally solving the stp by enforcing partial path consistency. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2008)*, pages 87–94, December 2008. ISBN 1368-5708.

Leon R. Planken, Mathijs M. de Weerdt, and Roman P.J. van der Krogt. P3c: A new algorithm for the simple temporal problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 256–263, Menlo Park, CA, USA, Sep 2008. AAAI Press. ISBN 978-1-57735-386-7.

Martha E. Pollack, David Joslin, and Massimo Paolucci. Flaw selection strategies for partial-order planning. *CoRR*, cs.AI/9706101, 1997.

Julie Porteous and Laura Sebastia. Extracting and ordering landmarks for planning. *Journal of Artificial Intelligence Research*, 22:2004, 2000.

Silvia Richter and Matthias Westphal. The LAMA planner. Using landmark counting in heuristic search. In *Proceedings of IPC-6*, 2008.

Silvia Richter, Malte Helmert, and Charles Gretton. A Stochastic Local Search Approach to Vertex Cover. In Joachim Hertzberg, Michael Beetz, and Roman Englert, editors, *KI 2007: Advances in Artificial Intelligence*, volume 4667 of *Lecture Notes in Computer Science*, pages 412–426. Springer Berlin Heidelberg, 2007.

Jussi Rintanen. An Iterative Algorithm for Synthesizing Invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth*

Conference on on Innovative Applications of Artificial Intelligence, AAAI 2000, pages 806–811, 2000.

Jussi Rintanen. Complexity of concurrent temporal planning. In *International Conference on Automated Planning and Scheduling*, pages 280–287. AAAI, 2007. ISBN 978-1-57735-344-7.

Earl D. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 206–214, 1975.

Bernd Schattenberg. *Hybrid Planning And Scheduling*. PhD thesis, Ulm University, Institute of Artificial Intelligence, 2009.

L.K. Schubert and A. Gerevini. *Accelerating Partial Order Planners by Improving Plan and Goal Choices*. Technical report. University of Rochester, Department of Computer Science, 1996.

Eddie Schwalb and Rina Dechter. Processing Disjunctions in Temporal Constraint Networks. *Artificial Intelligence*, 93:29–61, 1997.

J. Seipp and M. Helmert. Fluent Merging for Classical Planning Problems. In *Proceedings of the ICAPS-2011 Workshop on Knowledge Engineering for Planning and Scheduling*, pages 47–53, 2011.

Vikas Shivashankar, Ron Alford, Ugur Kuter, and Dana Nau. The GoDeL planning system: A more perfect union of Domain-Independent and hierarchical planning. 2013. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.362.8182`.

Reid G. Simmons and Håkan L. S. Younes. Vhpop: Versatile heuristic partial order planner. *CoRR*, abs/1106.4868, 2011.

David E. Smith, Jeremy Frank, and Ari K Jónsson. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review, 15(1):61–94*, 2000.

David E. Smith, Jeremy Frank, and William Cushing. The ANML language. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, September 2008.

Stephen F. Smith and Marcel A. Becker. An ontology for constructing scheduling systems. In *Working Notes from 1997 AAAI Spring Symposium on Ontological Engineering*, pages 120–129. AAAI Press, 1997.

Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

Austin Tate. Generating project networks. In R. Reddy, editor, *International Joint Conferences on Artificial Intelligence*, pages 888–893. William Kaufmann, 1977.

Edward P. K. Tsang. Constraint based scheduling: Applying constraint programming to scheduling problems. *J. Scheduling*, 6(4):413–414, 2003.

Thierry Vidal and Malik Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In Wolfgang Wahlster, editor, *ECAI*, pages 48–54. John Wiley and Sons, Chichester, 1996.

Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170(3): 298–335, March 2006. ISSN 0004-3702.

David E. Wilkins. Can ai planners solve practical problems? *Computational Intelligence*, 6(4):232–246, January 1991. ISSN 0824-7935. doi: 10.1111/j.1467-8640.1990.tb00297.x.

Lin Xu and Berthe Y. Choueiry. A new efficient algorithm for solving the simple temporal problem. In *Proceedings of International Symposium on Temporal Representation and Reasoning*, 2003.

HLS Younes and Reid Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20(1):405–430, 2003.

Appendix A: ANML Domains

Rovers

```
type Waypoint;
type Camera with {
     constant boolean on_board(Rover r);
     constant boolean supports(Mode m);
     predicate calibrated(Rover r);
};
type Objective;

type Rover with {
     variable Waypoint at;
     constant boolean equipped_for_soil_analysis;
     constant boolean equipped_for_rock_analysis;
     constant boolean equipped_for_imaging;
     constant boolean can_traverse(Waypoint x, Waypoint z);
     predicate have_soil_analysis(Waypoint p);
     predicate have_rock_analysis(Waypoint p);
     predicate have_image(Objective o, Mode m);
     variable boolean empty;
     variable boolean available;
};

type Store with {
     variable boolean empty;
     constant Rover store_of;
};

type Lander with {
     variable Waypoint at_lander;
     variable boolean channel_free;
};

type Mode;

predicate visible(Waypoint x, Waypoint y);
predicate visible_from(Objective o, Waypoint p);
predicate calibration_target(Camera c, Objective o);
predicate at_soil_sample(Waypoint p);
predicate at_rock_sample(Waypoint p);

predicate communicated_soil_data(Waypoint p);
predicate communicated_rock_data(Waypoint p);
predicate communicated_image_data(Objective o, Mode m);
```

```
action navigate(Rover x, Waypoint y, Waypoint z) {
       [all] {
             x.can_traverse(y, z) == true;
             x.available == true;
             visible(y, z) == true;
             x.at == y :-> z;
       };
};

action sample_soil(Rover x, Store s, Waypoint p) {
[all] {
      x.at == p;
      at_soil_sample(p) == true :-> false;
      x.equipped_for_soil_analysis == true;
      s.store_of == x;
      s.empty == true :-> false;
      x.have_soil_analysis(p) := true;
};
};

action sample_rock(Rover x, Store s, Waypoint p) {
[all] {
      x.at == p;
      at_rock_sample(p) == true :-> false;
      x.equipped_for_rock_analysis == true;
      s.store_of == x;
      s.empty == true :-> false;
      x.have_rock_analysis(p) := true;
};
};

action drop(Rover x, Store y) {
[all] {
      y.store_of == x;
      y.empty == false :-> true;
};
};

action calibrate(Rover r, Camera i, Objective t, Waypoint w) {
[all] {
      Camera.on_board(i, r) == true;
      r.equipped_for_imaging == true;
      calibration_target(i, t) == true;
      r.at == w;
      visible_from(t, w) == true;
      Camera.calibrated(i, r) := true;
};
```

```
};

action take_image(Rover r, Waypoint p, Objective o,
Camera i, Mode m) {
[all] {
      Camera.calibrated(i, r) == true :-> false;
      i.on_board(r) == true;
      r.equipped_for_imaging() == true;
      i.supports(m) == true;
      visible_from(o, p) == true;
      r.at() == p;
      r.have_image(o, m) := true;
};
};

action communicate_soil_data(Rover r, Lander l,
Waypoint p, Waypoint x, Waypoint y) {
[all] {
      l.channel_free() == true :-> true;
      r.available() == true :-> true;
};
[all] {
      r.at == x;
      l.at_lander == y;
      r.have_soil_analysis(p) == true;
      visible(x, y) == true;
      communicated_soil_data(p) := true;
};
};

action communicate_image_data(Rover r, Lander l,
Objective o, Mode m, Waypoint x, Waypoint y) {
[all] {
      l.channel_free() == true :-> true;
      r.available() == true :-> true;
};
[all] {
      r.at == x;
      l.at_lander == y;
      r.have_image(o, m) == true;
      visible(x, y) == true;
      communicated_image_data(o, m) := true;
};
};

action communicate_rock_data(Rover r, Lander l,
Waypoint p, Waypoint x, Waypoint y) {
[all] {
```

```
       l.channel_free() == true :-> true;
       r.available() == true :-> true;
};
[all] {
      r.at == x;
      l.at_lander == y;
      r.have_rock_analysis(p) == true;
      visible(x, y) == true;
      communicated_rock_data(p) := true;
};
};
```

Elevators

```
type Floor < object;

type Passenger < object with {
  variable Floor location;
};

type Elevator < object with {
  variable Floor location;
  reusable integer [0,infinity] occupancy;
};

producible integer totalCost();
constant boolean connected(Floor a, Floor b);
constant boolean reachable(Elevator e, Floor a);

action Move(Elevator e, Floor a, Floor b) {
  end = start + 10;
  [all] {
    reachable(e,a) == true;
    reachable(e,b) == true;
    connected(a,b) == true;
    e.location == a :-> b;
    totalCost :produce 1;
  };
};

action Transport(Passenger p, Elevator e, Floor a, Floor b) {
  [all] {
    reachable(e,a) == true;
    reachable(e,b) == true;
    connected(a,b) == true;
    e.occupancy :use 1;
    totalCost :produce 1;
```

```
    p.location == a :-> b;
  };
  [start,start+1] e.location == a;
  [end-1,end] e.location == b;
};
```

VisitAll

```
type Location;
variable Location at;
function boolean visited(Location a);
constant boolean connected(Location a, Location b);

action move(Location a, Location b){
     [all] {
        connected(a,b) == true;
        at == a :-> b;
        visited(b) := true;
     };
};
```

Handover

```
type Location;

type NavLocation < Location;

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

type Item with {
  variable Location location;
};

constant boolean navigable(NavLocation a, NavLocation b);

constant boolean passable(NavLocation a, NavLocation b);

action Move(Robot r, NavLocation a, NavLocation b) {
  [all] {
    r.location == a :-> b;
    navigable(a, b) == true;
  };
};
```

```
action Handover(Robot r1, Robot r2,
Item i, NavLocation a, NavLocation b) {
  [all] {
    passable(a, b) == true;
    r1.location == a;
    r2.location == b;
    i.location == r1 :-> r2;
  };
};

action Pick(Robot r1, Item i, NavLocation a) {
  [all] {
    r1.location == a;
    i.location == a :-> r1;
  };
};

action Transport(Robot r1, Robot r2,
NavLocation l1, NavLocation l2, Item i) {
  :decomposition{ ordered(
     Pick(r1, i, l1),
     Pass(r1, r2, i),
     Drop(r2, i, l2));
  };
};

action Pass(Robot r1, Robot r2, Item i) {
  :decomposition{
    constant NavLocation l1;
    constant NavLocation l2;
    Handover(r1, r2, i, l1, l2);
  };

  :decomposition{
    constant NavLocation l1;
    constant NavLocation l2;
    constant NavLocation l3;
    constant NavLocation l4;
    constant Robot r3;
    ordered(
      pass : Pass(r1, r3, i),
      handover : Handover(r3, r2, i, l1, l2));
    end(pass) < start(handover);
  };
};

action Drop(Robot r1, Item i, NavLocation a) {
```

```
  [all] {
    r1.location == a;
    i.location == r1 :-> a;
  };
};
```

Appendix B: Attached CD

The attached CD contains the translation tool described in Section 1.5 and the planning system FAPE, including all its components. The directories are structured as follows.

```
/
├── thesis.pdf
├── FAPE
│   ├── readme.txt
│   ├── test.bat
│   ├── test.sh
│   ├── executable
│   ├── example
│   ├── source
│   ├── experiments
│   └── problems
└── translator
    ├── readme.txt
    ├── test.bat
    ├── test.sh
    ├── executable
    ├── example
    ├── source
    ├── experiments
    └── problems
```

The `readme` files contain instructions and examples on how to run the executables, both require Java Virtual Machine 1.7+. The `problems` folders contain the data sets used for experiments and in case of FAPE, also several other ANML testing domains and problems. FAPE's executable is compiled as a planning system, running FAPE as an acting system requires to implement the listener of the acting component according to a particular integration scenario and we suggest that the reader contacts the author.

The scripts in files `test.bat` and `test.sh` contain example runs of the executables and should be directly executable on their corresponding platforms (Windows/Unix). Folder `example` contains the output of an example run, in case of FAPE also visualizations of the produced plan and the temporal network. Folder `experiments` contains the data collected during experiments, including visualizations that did not fit into the thesis.