IAAS Service Computing Department

Masterarbeit

A Proof of Concept for Risk-aware Plan-based HTN Planning

Tobias Hirzel

Course of Study:	Software-Engineering
Examiner:	Dr. Ilche Georgievski
Supervisor:	Ebaa Alnazer, M.Sc.
Commenced:	April 26, 2023
Completed:	October 26, 2023

Abstract

With the rise of automation comes the need for automated decision-making. Real-world domains of business and industry often utilize automated agents, like robots, to perform various tasks. Automated agents rely on decision-making software to decide on a course of action to complete such tasks. When decision-making software chooses a sequence of actions for an automated agent to perform, a plan is created. This process of planning as well as decision-making software are subjects of study in the research field of Automated Planning, where a decision-making software is referred to as *AI Planner*. The manner in which an AI Planner conducts planning depends on the planning technique employed. One of the more widely used planning techniques is Hierarchical Task Network (HTN) planning, which composes available actions of a domain into a hierarchy, to allow for planning, resembling the way a human individual would conduct planning. A crucial factor in human planning is the notion of risk, since, in real-world domains, actions frequently lead to one of many possible results. Often there is no guarantee for a particular result, and some results might be undesirable. Therefore, when an individual performs planning, they take risk into account, as the individual stands to lose a certain amount of resource, such as money or time. The way an individual approaches planning with risk in mind, shows an attitude towards risk. An individual, avoiding risk bearing actions, shows a risk averse attitude, while an individual, taking risks, shows a risk seeking attitude. If no risk consideration is made, the individual shows a risk neutral attitude. For particular real-world domains, a reasonable individual would show a specific risk attitude out of necessity. Consider aviation, for example, where many would agree, that risk avoiding actions should be chosen. When risk bearing actions are chosen in such high stakes domains, undesirable results can include the loss of human lives. Therefore, dealing with risk by showing the appropriate risk attitude, is especially important in such domains. Since automated agents are often employed in those domains, AI Planners should be aware of risk and factor risk into planning. Furthermore, AI Planners should be able to perform planning according to an appropriate risk attitude, so automated agents can act according to this risk attitude. However, as far as we know, a risk-aware AI Planner is a rarity in the research field of Automated Planning. To contribute towards risk-aware planning, we present a proof of concept for an approach to risk-aware HTN planning. The approach employs utility functions to heuristically guide planning towards plans, that adhere to a specified risk attitude. The proof of concept comes in the form of Risk Aware PANDA3 (RAPANDA3), an extension to the AI Planner: Planning and Acting in a Network Decomposition Architecture 3 (PANDA3). On top of that, we propose an addition to an HTN input language, which allows the modelling of risk for actions. Furthermore, we provide multiple self designed HTN domains, which use this addition. With these domains, we conduct an evaluation of RAPANDA3, to scrutinize the implemented approach to risk-aware HTN planning, and to show, that this approach results in plans, adhering to a specified risk attitude.

Kurzfassung

Mit zunehmender Automatisierung steigt die Notwendigkeit einer automatisierten Entscheidungsfindung. In realen Domänen (domains) des Geschäfts- und Industriebereichs werden häufig automatisierte Akteure (agents) wie Roboter eingesetzt, um verschiedenste Aufgaben auszuführen. Automatisierte Akteure verlassen sich auf Entscheidungssoftware, um über eine Vorgehensweise zur Erledigung solcher Aufgaben zu entscheiden. Wenn Entscheidungssoftware eine Abfolge von Aktionen (actions) auswählt, die ein automatisierter Akteur ausführen soll, erstellt sie einen Plan. Dieser Prozess der Planung (planning) sowie Entscheidungssoftware sind Gegenstand der Untersuchung im Forschungsfeld des Automated Planning, in dem Entscheidungssoftware als *AI Planner* bezeichnet wird. Die Art und Weise, wie ein AI Planner die Planung durchführt, hängt von der verwendeten Planungstechnik (planning technique) ab. Eine der am weitesten verbreiteten Planungstechniken ist die Hierarchical Task Network (HTN) Planung, welche verfügbare Aktionen einer Domäne in einer Hierarchie zusammenfasst, um eine Planung zu ermöglichen, die der Art und Weise ähnelt, wie ein menschliches Individuum eine Planung durchführen würde. Ein entscheidender Faktor bei der menschlichen Planung ist Risiko, da Aktionen in der realen Welt häufig zu einem von vielen möglichen Ergebnissen führen. Oft gibt es keine Garantie für ein bestimmtes Ergebnis und manche Ergebnisse können unerwünscht sein. Daher wird Risiko von einer Person berücksichtigt, die sich im Planungsprozess befindet, da die Person Gefahr läuft, eine bestimmte Menge an Ressourcen wie Geld oder Zeit zu verlieren. Die Art und Weise, wie eine Person Risiko in ihre Planung einfließen lässt, zeigt eine Einstellung zum Risiko. Ein Individuum, das risikobehaftete Aktionen vermeidet, zeigt eine risikoscheue Einstellung (risk averse attitude), während ein Individuum, das Risiken eingeht, eine risikofreudige Einstellung (risk seeking attitude) zeigt. Erfolgt keine Risikoabwägung, zeigt das Individuum eine risikoneutrale Einstellung (risk neutral attitude). In bestimmten Domänen der realen Welt zeigt ein vernünftiges Individuum eine bestimmte Risikoeinstellung aus Notwendigkeit. Zum Beispiel in der Domäne der Luftfahrt, wo viele zustimmen würden, dass Risiko vermeidende Aktionen gewählt werden sollten. Wenn risikobehaftete Aktionen in solch Hochrisiko Domänen gewählt werden, können die möglichen unerwünschten Ergebnisse den Verlust von Menschenleben nach sich ziehen. Dadurch ist in solchen Domänen der Umgang mit Risiko durch eine passende Risikoeinstellung besonders wichtig. Da in diesen Domänen häufig automatisierte Akteure eingesetzt werden, sollten Ai Planner Risiko bewusst planen. Darüber hinaus sollten AI Planner in der Lage sein, die Planung entsprechend einer angemessenen Risikoeinstellung durchzuführen, sodass automatisierte Akteure gemäß dieser Risikoeinstellung handeln können. Jedoch, soweit wir wissen, ist ein risikobewusster AI Planner im Forschungsfeld des Automated Planning eine Seltenheit. Um zu risikobewusster Planung beizutragen, präsentieren wir einen Proof of Concept für ein Vorgehen der risikobewussten HTN Planung (risk-aware HTN planning). Das Vorgehen verwendet Nutzenfunktionen (utility functions), um Planung heuristisch auf Pläne auszurichten, die einer bestimmten Risikoeinstellung entsprechen. Der Proof of Concept kommt in der Form von Risk Aware PANDA3 (RAPANDA3), eine Erweiterung des AI Planner: Planning and Acting in a Network Decomposition Architecture 3 (PANDA3). Darüber hinaus schlagen wir eine Ergänzung zu einer HTN Eingabesprache vor, welche die Modellierung von Risiken für Aktionen ermöglicht. Weiterhin stellen wir mehrere selbst gestaltete HTN Domänen zur Verfügung, die diese Ergänzung nutzen. Mit diesen Domänen führen wir eine Evaluierung von RAPANDA3 durch, um das Vorgehen der risikobewussten HTN Planung zu überprüfen und zu zeigen, dass dieses Vorgehen zu Plänen führt, die einer bestimmten Risikohaltung entsprechen.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

ANTLR Another Tool for Language Recognition. 41

DNBs dynamic Bayesian networks. 76

FAPE Flexible Acting and Planning Environment. 18

GPS General Problem Solver. 21

HDDL Hierarchical Domain Definition Language. 11

HTN Hierarchical Task Network. 18

IPC International Planning Competition. 51

LCFR Least-Cost Flaw-Repair. 46

MDP Markov decision process. 75

NOAH Nets of Action Hierarchies. 21

PANDA3 Planning and Acting in a Network Decomposition Architecture 3. 18

RAPANDA3 Risk Aware PANDA3. 18

SHOP2 Simple Hierarchical Ordered Planner. 18

SHPE Simple Hierarchical Planning Engine. 18

STRIPS Stanford Research Institute Problem Solver. 21

TDG Task Decomposition Graph. 25

1 Introduction

Risk is ubiquitous in real-world domains. Due to its non-deterministic nature, reality is laden with risk and particular outcomes are rarely guaranteed. Therefore, many *actions* are risk bearing, as they might not yield the desired outcome. An everyday example is the action of buying over the internet, which could save time but risks receiving a product, that is not as advertised. Any action is performed with a goal in mind, and often the goal requires multiple actions to reach. When multiple actions are chained together, for the purpose of reaching a goal, a *plan* is formed. Since many actions are risk bearing, plans are similarly risk laden and in most cases the individual, conducting the planning, stands to lose a certain amount of resource, such as money. Resource losses can amount to significant values in high stakes domains, therefore, dealing with risk while planning is prudent. Domains in business and Industry are often considered high stakes, and frequently rely on automated *agents*, such as robots or autonomous cars. Automated agents, similar to human agents, require plans to fulfil goals. Therefore, automated agents are in need of automated planning. As risk is an important factor for planning in high stakes domains, automated planning should consider risk.

Planning for automated agents is a subject of study in the research field of *Automated Planning*, where *planners* are employed to conduct *planning* for automated agents. A plan is a sequence of actions, performable in the real-world domain the automated agent acts in. A planner receives a formal representation of the real-world domain, presenting the set of performable actions, with which the planner can decide on a plan. Since planners are tasked with choosing actions for automated agents in high stakes domains, they should be aware of risk and factor risk into decision-making. However, we found it to be rare to come across a planner that is risk-aware. Moreover, the formal representations of the domains to plan in, often do not have the necessary formalisms in place to allow for risk-aware planning. Therefore, we present a risk-aware planner as proof of concept for an approach to risk-aware planning. On top of that, we propose a formalism that allows domain representations to accommodate risk awareness.

To achieve risk-aware planning, we observe human behaviour when confronted with risk and recognize three *risk attitudes*: risk neutral, risk seeking and risk averse. When faced with a decision between multiple actions, a risk neutral agent chooses the action, with the lowest expected cost. While this can be a sensible way to deal with risk, human agents rarely act this way, and for good reason. Depending on the personality and the domain they act in, human agents tend to embrace or avoid risk. In Space programs, buying cheaper, low quality rocket parts might reduce expected cost for a rocket. However, it seems much more reasonable to avoid low quality parts, as human lives and a loss of public trust are at stake. If the potential loss is catastrophic, human agents most likely show a risk averse attitude. In contrast, human agents in a casino could conceivably exhibit a risk seeking attitude, enjoying the thrill of betting a large sum on a single die roll.

Utility functions are commonly employed to model risk attitudes mathematically, since they provide a mapping from objective cost to subjective cost [ZZY+14]. In Alnazer, Georgievski, and Aiello [AGA22b], the authors propose to introduce risk attitudes, and therefore risk awareness, to planning by usage of utility functions. We follow up on this, by providing a means with which a planner can make use of utility functions to create plans expressing a risk attitude. This comes in the form of a heuristic that guides planning to adhere to a risk attitude.

The risk-aware planning approach, we describe in this work, is an extension to the framework of Hierarchical Task Network (HTN) planning, a widely used Automated Planning technique. HTN planning is utilized in multiple fields of science and industry, such as web service composition, medicine, robotics, video games and smart buildings [GNN+17; GTFM13; KBK+07; KBK08; MAMO18; SPW+04]. HTN planning uses a hierarchical structure to formalize real-world domains, which is intuitive for users taking part in the planning process, as it closely resembles the way humans organize problems. Furthermore, the hierarchy uniquely allows for the expression of useful constraints on what plans can be considered solutions [BBHB17]. There are many already established planner that implement HTN planning, such as Simple Hierarchical Ordered Planner (SHOP2), Simple Hierarchical Planning Engine (SHPE), Flexible Acting and Planning Environment (FAPE) and Planning and Acting in a Network Decomposition Architecture 3 (PANDA3) [DBIG14; HBBB21; MJC14; NAI+03]. To realize our proof of concept of risk-aware HTN planning, we modify PANDA3 to obtain Risk Aware PANDA3 (RAPANDA3).

The contributions of our work are the following:

- We present risk-aware plan-based HTN planning, an approach to HTN planning that employs utility functions to guide planning towards plans that adhere to a risk attitude.
- We provide an in-depth look at the PANDA3 source code and by which means PANDA3 conducts plan-based planning.
- We introduce RAPANDA3, a proof of concept for risk-aware plan-based HTN planning, build on top of plan-based HTN planning of PANDA3.
- We propose an extension of one HTN input language to accommodate risk awareness. Even though the input language we extend is a general input language for HTN planning, the extended parser is integrated with RAPANDA3. Therefore, this contribution is specific to RAPANDA3. However, it is easily portable to similar parser used in other planners.
- We provide several HTN Domains, created with risk awareness in mind.
- An evaluation of RAPANDA3 is conducted, against the newly created HTN domains, to examine risk-aware plan-based HTN planning. The results of this evaluation are presented and discussed.

Following the introduction, Chapter 2 elaborates on planning in general and plan-based HTN planning in detail. Additionally, we discuss heuristics and how heuristics are used to inform planning. Chapter 3 contains our description of risk-aware plan-based HTN planning. In Chapter 4, we present an in-depth look at how PANDA3 conducts plan-based HTN planning, by means of UML diagrams and description. Following this, we elaborate on the changes made to obtain RAPANDA3. Chapter 5 introduces our benchmark domains, describes our evaluation of RAPANDA3, and discusses the results of the evaluation. Chapter 6 houses a discussion about scientific works, related to our work. Finally, in Chapter 7, we provide a summary of our findings and present suggestions for next steps.

2 Background

Chapter 2 starts with a brief history overview of the Automated Planning research field. This includes a short introduction to *Classical Planning* and HTN Planning. Afterwards, HTN Planning is elaborated on by defining several relevant terms and concepts. We additionally introduce HDDL, an input language for HTN Planning. On top of that, simple examples of HDDL files are presented to illustrate the HTN method and HDDL language. Following after, is an explanation of the role of heuristics in HTN Planning and a description of a cost aware heuristic.

2.1 Automated Planning

The research field of Automated Planning has its roots in psychology, with the first ever introduced planner being *General Problem Solver (GPS)*, aiming to simulate human thought processes [NS61]. From there, various other planners followed, such as *Stanford Research Institute Problem Solver (STRIPS)* [FN93] and *WARPLAN* [War76], all building on top of a similar technique to planning. This technique was later coined *Classical Planning*, when Automated Planning became more diverse in its approaches. The notion of hierarchy was first introduced to Automated Planning with *Nets of Action Hierarchies (NOAH)*[GA15], the first planner to implement Hierarchical Planning. While there are multiple approaches for incorporating hierarchies into planning [BAH19], our work is only concerned with HTN Planning.

Planning entails finding a sequence of actions to achieve a desired goal, limited to the possible actions, set by the environment, the planning is conducted in. Classical Planning captures this by defining a *state* and a set of *actions* that manipulate the state. Moreover, the environment is depicted by a *domain*, which holds the possible actions. A *problem instance* sets forth an initial state and a set of goal states. The planning challenge consists of finding a sequence of actions, that is applicable to the initial state and leads to one of the goal states. For an action to be applicable, its *preconditions* need to be met by the state. Additionally, actions exert *effects*, which leave the state modified after execution of the action. In consequence, every action can be viewed as a state transition between a precondition fulfilling state and an effects bearing state. A chain of state transitions constitutes a *plan*. If this chain reaches a goal state from the initial state, it is called a *solution* to the given problem instance. While HTN Planning approaches can differ from approaches of Classical Planning significantly, having an understanding of the just roughly described method, is helpful to grasp HTN Planning approaches.

2.2 HTN Planning

Planning is conducted in a certain environment, formally defined by a domain D. A problem instance PI, provides the challenge a plan has to be created for and is closely associated with a domain. We provide the *satellite_simple* domain in Listing 2.1, with a corresponding problem instance in Listing 2.2. The format for these examples is HDDL, an input language recently proposed for HTN Planning [HBB+19]. HDDL consists of *sections*. A section is prefixed with a colon and section name and holds entities[1], performing the role for the domain or problem instance, the section name suggests. Entities a section holds are bordered with parentheses, with the opening parenthesis starting before the section colon. For instance, the *(:types satellite instrument)* section includes the entities "satellite" and "instrument", indicating that available types in the domain are *satellite* and *instrument*. Sections can encapsulate subsections, in this case the subsection specifies roles for the parent section. For instance, *(:task loadInstr :parameters (?i - instrument ?s - satellite))* specifies that, *?i - instrument* and *?s - satellite* are parameters of *loadInstr*. For the rest of this work, we use HDDL for domain and problem instance description. The examples are provided to support the explanations in this chapter while simultaneously introducing the HDDL language.

Hierarchy composition In HTN planning, domain actions are often called *tasks* and for the sake of consistency, we will use this term from here on out. Domain tasks have an underlying hierarchy, which is composed by an interplay between *compound tasks*, *primitive tasks* and *methods*.

Definition 2.2.1 (Task)
A Task t represents an action that can be performed in a Domain D. A Task can either be primitive or compound. We denote primitive tasks with tp and compound tasks with tc.

Definition 2.2.2 (Task network)
Any finite sequence of tasks is called a task network Tn.

Primitive tasks represent the leaves of the domain hierarchy and the atomic building block of a plan P. Primitive tasks can be executed directly, and a plan only consists of primitive tasks. We call the primitive tasks in a plan, *plan steps*.

Definition 2.2.3 (Plan)
A plan P in a domain D is a task network Tn comprised solely of primitive tasks, which are performable in the domain D.

Our sample domain, in Listing 2.1, supplies three primitive tasks, *getInstr*, *initializeInstr* and *calibrateInstr*. A primitive task is specified with the *:action* section in HDDL. In contrast to primitive tasks, Compound tasks are at least one level above the lowest level in the hierarchy and cannot be executed directly.

Definition 2.2.4 (Partial plan)
A task network containing at least one compound task is not executable, we call such a task network a partial plan PAR.

[1]We call them entities for the sake of explanation, however that is not an official HDDL term.

Listing 2.1 This listing contains *satellite_simple*, an example domain in HDDL format. It depicts satellites getting instruments loaded and initialized. A small part is omitted for the sake of brevity.

```
(define (domain satellite_simple)

;---- omission

;---------------- types and predicates -----------------------
(:types satellite instrument)

(:predicates
(onBoard ?i - instrument ?s - satellite) (initialized ?i - instrument) (calibrated ?i -
instrument)
)

;---------------- compound tasks -----------------------
(:task loadInstr
  :parameters (?i - instrument ?s - satellite) )

(:task prepareInstr
  :parameters (?i - instrument ?s - satellite) )

;---------------- methods -----------------------
(:method m_loadInstr
  :parameters (?i - instrument ?s - satellite)
  :task (loadInstr ?i ?s)
  :precondition()
  :ordered-subtasks (and (getInstr ?i ?s)  (prepareInstr ?i)))

(:method m_prepareInstr
  :parameters (?i - instrument ?s - satellite)
  :task (prepareInstr ?i ?s)
  :precondition()
  :ordered-subtasks (and (initializeInstr ?i ?s)  (calibrateInstr ?i ?s)))

;---------------- primitive tasks -----------------------
(:action getInstr
  :parameters (?i - instrument ?s - satellite)
  :precondition (not (onBoard ?i ?s ))
  :effect (onBoard ?i ?s))

(:action initializeInstr
  :parameters (?i - instrument ?s - satellite)
  :precondition (and (not (initialized ?i)) (onBoard ?i ?s))
  :effect (initialized ?i))

(:action calibrateInstr
  :parameters (?i - instrument ?s - satellite)
  :precondition (and (not (calibrated ?i)) (onBoard ?i ?s))
  :effect (calibrated ?i))
)
```

Listing 2.2 The sample problem instance *p_example* for the *satellite_simple* domain, shown in Listing 2.1.

```
(define (problem p_example)
  (:domain satellite_simple)

;---------------- constants -----------------------
  (:objects
    satellite0 - satellite
    instrument0 - instrument
    satellite1 - satellite
    instrument1 - instrument
  )

;--------------- initial task network -----------------
  (:htn :parameters () :ordered-subtasks
    (task1 (loadInstr instrument1 satellite1))
  )
;--------------- initial State -----------------
  (:init
    (onBoard instrument0 satellite0)
  )
)
```

Two compound tasks are supplied in *satellite_simple*, namely *loadInstr* and *prepareInstr*. While compound tasks can not be carried out directly, they are the target for decomposition by *methods*.

Definition 2.2.5 (Method)
A method M consists of a pair $((tc(M)), (Tn(M)))$, where $tc(M)$ denotes the compound task that can be decomposed and $Tn(M)$ specifies a task network which we call subtasks of M.

We have two methods in *satellite_simple*, called *m_loadInstr* and *m_prepareInstr*. A method can only decompose one compound task, yet a compound task can be the target for multiple methods. The decomposition target of a method can be identified by looking at the *:task* section of the method in question. In the *satellite_simple* domain, *m_loadInstr* can decompose *loadInstr* for example. Methods posses *subtasks*, which is a task network that can contain primitive or compound tasks. By decomposing a compound task in a task network, the compound task is substituted by the subtasks of the decomposing method.

Definition 2.2.6 (Decomposition)
A method M decomposes tc in a task network Tn, if $tc = tc(M)$. Decomposing a compound task tc with M replaces tc with the subtasks $Tn(M)$ in Tn, which results in Tn_{new}. We call Tn_{new} an evolution of Tn. For decomposition, we also say: a compound task tc decomposes into the subtasks $Tn(M)$, for method M.

Which subtasks a method possesses are denoted by the *:ordered-subtasks* section. For example, in *satellite_simple*, the method *m_loadInstr* has the tasks *getInstr* and *prepareInstr* as subtasks. The section is called *:ordered-subtasks*, as there are three approaches to decomposition. Totally ordered, partially ordered and unordered decomposition. When replacing a compound task in a task network with subtasks, the totally ordered approach preserves the subtasks' order and the subtasks are placed at the position of the replaced compound task, with respect to the rest of the tasks in the task network. HTN Planning using this approach is often labelled as *totally ordered HTN Planning* and for the rest of our work, all decompositions are assumed to utilize the totally ordered approach.[GA15].

Definition 2.2.7 (Totally ordered HTN Planning)
HTN planning is totally ordered, if every decomposition in every given task network preserves the order of the subtasks $Tn(M)$ and the order of all other tasks in the task network.

Task Decomposition Graph A common graphical representation of the underlying hierarchy in a domain is the so called Task Decomposition Graph (TDG), which shows a task network's possible decompositions [EBSB12]. Creating a TDG for our example domain in Listing 2.1 yields the graph depicted in Figure 2.1 and provides an overview over the underlying hierarchy. The highest level compound task is *loadInstr*, which is decomposed by the method *m_loadInstr*. The ordered subtasks of method *m_loadInstr* are *getInstr* and *prepareInstr*. Since *getInstr* is primitive, there is no method targeting it. However, *prepareInstr* is a compound task, and thus it can be decomposed by a method, in this case *m_prepareInstr*. The method *m_prepareInstr* decomposes into the primitive tasks *initializeInstr* and *calibrateInstr*. A more formal definition is provided in Definition 2.2.8, which is adapted from [BBHB17].

Definition 2.2.8 (Task Decomposition Graph (TDG))
A TDG is a bipartite graph $G = (V_T, V_M, E_{T\to M}, E_{M\to T})$, with V_T representing task vertices, $v_t \in V_T$, V_M representing method vertices $v_m \in V_M$, $E_{T\to M}$ representing edges from task vertices to method vertices, $(v_t, v_m) \in E_{T\to M}$, and $E_{M\to T}$ representing edges from method vertices to task vertices, $(v_m, v_t) \in E_{M\to T}$. A method vertex represents the method it is named after, and a task vertex represent the task it is named after.

State and predicates A crucial part of planning is to bring *constants* into a desired condition. The condition of a constant is stored in the *state S*. The state consists of multiple *predicates*, with one predicate p describing a condition of a constant or a relation between constants. We denote the set of all predicates as Q. Predicates can either be false or true, and if a predicate does not exist in the state, it is automatically false. A *type* is a variable constant, with a constant being a concretization of one type. Consequently, every constant is of one type, however a constant can have multiple types if the concretized type extends another type, e.g. *car* extends *vehicle*. The example domain and problem instance, given in Listing 2.1 and Listing 2.2, exemplify this relationship between types, constants and predicates. Our example domain *satellite_simple* provides two types, *satellite* and *instrument*, through the *:types* section. In our problem instance *p_example*, one can identify four constants, with their types suffixed. One example is *satellite0 - satellite*, which defines a constant, identified as *satellite0*, being of the type *satellite*. Looking back to the *satellite_simple* domain, several predicates are given through the *:predicates* section and the first predicate is *(onBoard ?i - instrument ?s - satellite)*. This predicate is in the so-called *lifted* notation, since it has at least one unbound *variable* as a *parameter variable*. Analogously to constants, unbound parameter variables are of a type. For the predicate *(onBoard ?i - instrument ?s - satellite)* the unbound parameter variables are *?i* of type *instrument* and *?s* of type *satellite*. Binding both those parameter variables to constants of the same type is called *instantiating* and allows for the inclusion of a predicate

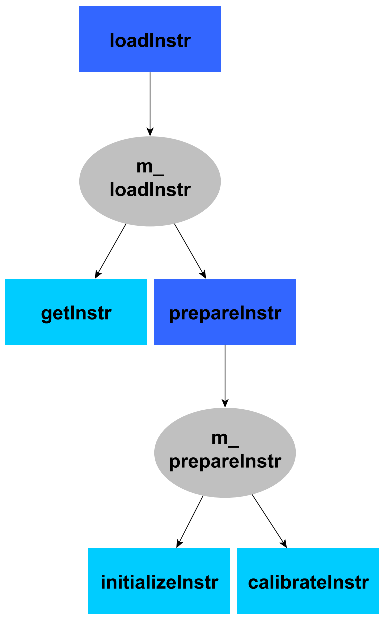


Figure 2.1: A diagram, depicting the TDG for our example domain *satellite_simple* listed in Listing 2.1. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks.

instance into the state. An instance is given, only if all parameter variables are bound. For example, the *initial state* of our example problem *p_example* consists of the predicate *(onBoard instrument0 satellite0)*, which is an instance of *(onBoard ?i - instrument ?s - satellite)*, describing a relation between concrete constant *instrument0* and concrete constant *satellite0*.

Preconditions and effects Predicates, and therefore the state, can be manipulated by the effects of *operators*. Operators are wrappers around primitive tasks, that define preconditions and effects.

Definition 2.2.9 (Operator)
An operator o is related to exactly one primitive task tp and defines its preconditions and effects with the triple $(tp(o), pre(o), eff(o))$ with $pre(o) \in 2^{Q}$ and $eff(o) \in 2^{Q}$.

We assume a primitive task having exactly one operator and so, for the sake of brevity, it is convenient to attribute the preconditions and effects of an operator to the primitive task the operator wraps, therefore we will mention operators explicitly only if necessary. Preconditions and effects are each a list of predicates. A primitive task is applicable if the preconditions match the predicates in the state. The effects of a primitive task are introduced into the state, when the primitive task is executed. Analogously to predicates, a primitive task has parameter variables, thus it can be instantiated as well. In HDDL, the *:parameters* section holds the parameter variables of a primitive task. When inspecting our example domain in Listing 2.1, the primitive task *getInstr* has *:parameters (?i - instrument ?s - satellite)* as parameter section, providing *?i - instrument* and *?s - satellite* as parameter variables. Furthermore, its effects are defined in the *:effect* section as *(onBoard ?i ?s)*, a version of the lifted predicate *(onBoard ?i - instrument ?s - satellite)* without type indication. With this, the effects of *getInstr* introduce an instance of *(onBoard ?i - instrument ?s - satellite)* to the state, with the parameter variables of *onBoard* bound to the constants the parameter variables in the *:parameters* section are bound to. This works equivalently for the preconditions of the primitive task *getInstr*, which are supplied by the *:precondition* section, except the predicate instances do not get introduced into the state, instead they need to be present already. Looking at our example domain *satellite_simple*, one can observe that methods and compound tasks posses parameter variables similar to primitive tasks. Consequently, methods and compound tasks can be instantiated or lifted as well. When decomposing a compound task instance, constants bound to parameter variables are used to instantiate the subtasks. If any structure consisting of tasks and methods only contains instances, we call it *ground* or *grounded*, for example a task network made up entirely of instances of tasks is a grounded task network.

Planning starts with the *initial task network* representing the problem instance's goal.

Definition 2.2.10 (Initial task network)
An initial task network Tn_0 is the entry point to planning for problem instance PI and domain D.

In our example problem instance listed in Listing 2.2, the initial task network is a partial plan consisting of the compound task instance: *(loadInstr instrument1 satellite1)*. A plan can now be computed by decomposing compound tasks of partial plans until only primitive tasks are present. A plan is a *solution* if its sequence of primitive tasks is applicable to the initial state, and every primitive task is applicable to the state created by its predecessor's execution. [GA15].

Definition 2.2.11 (Solution)
A plan P is a solution S to a problem instance PI, if it depicts an ordered sequence of primitive tasks, that is decomposable from the problem instance's initial task network and is applicable to the initial state of the problem instance and every primitive task is applicable to the state created by its predecessor's execution.

Plan-based HTN planning HTN planning can roughly be divided into two approaches, *plan-based HTN planning* and *state based HTN planning*. They differ in planning execution and the information stored in the *search space*. The term search space describes temporary information necessary for planning. In state based HTN planning, the search space is a subset of the state space, induced by the domain and problem instance. It contains snapshots of the state for different planning phases. Whenever a decomposition results in a primitive task, this task is added to the plan and the planning progresses into a new phase, by applying the effects of the just added task to the old state snapshot and continuing on with the newly formed state snapshot [GA15]. In contrast, the search space in plan-based HTN planning contains plans and partial plans, thus we will call it *plan space* from here on out.

Definition 2.2.12 (Plan space)
A plan space PS is a temporary data structure, persisting through one plan-based HTN planning. PS contains all task networks that have not been evolved (term is explained shortly) yet. Additionally, PS contains task networks that are plans and not checked for being a solution yet.

An outline of the general approach of plan-based HTN planning is provided in Algorithm 2.1. This approach is explained in the following. At the start of planning, only the initial task network is in the plan space. From there, the algorithm processes task networks iteratively. From the plan space, a task network is chosen to be evolved and deleted from the plan space. The task network to

Algorithm 2.1 The algorithm of plan-based HTN planning, details could be different for individual implementations.

```
procedure PLAN-BASED HTN PLANNING($D$,$PI$)
    PS ← $Tn_0 \in PI$
    while PS != ∅ do
        $Tn_{evolving}$ ← chooseAndDelTn(PS)
        if isSolution($Tn_{evolving}$) then
            return $Tn_{evolving}$
        end if
        compTaskToDecompose ← chooseCompTask($Tn_{evolving}$)
        for all (method M ∈ D with tc(M) = compTaskToDecompose do
            $Tn_{new}$ ← M.decompose($Tn_{evolving}$, compTaskToDecompose)
            PS.add($Tn_{new}$)
        end for
    end while
    return unsolvable
end procedure
```

evolve is checked for being a solution. If it is a solution, without the loss of generality, we assume the task network to evolve is returned and planning stops. If the evolving task network is not a solution, a compound task is chosen. How this compound task is chosen depends on the specific implementation of plan-based HTN planning. For every method targeting this chosen compound task, a decomposition is conducted. One decomposition evolves the task network to evolve, by substitution of the chosen compound task, with the subtasks of the decomposing method. This newly evolved task network is added to the plan space. Now, the next task network in the plan space is assigned to be evolved, and the loop starts anew. We provide no algorithm for state-based HTN planning, since our work revolves around plan-based HTN planning.

2.3 Heuristics

Our sample domain in Listing 2.1 is simple enough to compute to only one possible solution for a given problem instance. However, more intricate domains might have several possible solutions, which beckons the question for a quality solution. For quality to exist, there must be at least one quality criterion. We propose the intuitive criterion *task cost*, or just *cost*, also denoted as $c(tp)$, which describes the cost of the execution of a primitive task measured in abstract *units*. The cost of a solution is the sum over all task costs, and the optimal solution is the one with the lowest cost.

A common way for several possible solutions to arise is having multiple different methods targeting the same compound task, since different methods decompose to different sets of subtasks. This challenges a planner to make a choice between those methods. However, making a choice between methods, in the context of plan-based HTN planning, does not equate to the planner actually choosing a method at the moment of decomposition. When looking at Algorithm 2.1, one can observe that the task network to evolve is evolved into several evolutions by all methods targeting the compound task. Nevertheless, a planner commits to a past method choice when it commits to a solution. That is because a solution represents a path of evolutions of task networks, starting from

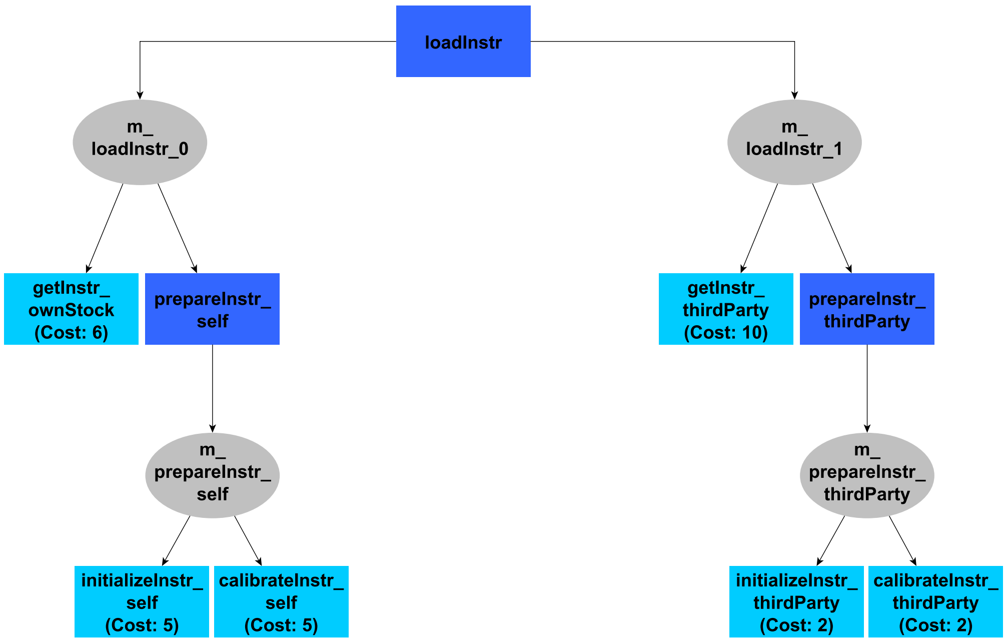

loadInstr
m_ loadInstr_0
m_ loadInstr_1
getInstr_ ownStock (Cost: 6)
prepareInstr_ self
getInstr_ thirdParty (Cost: 10)
prepareInstr_ thirdParty
m_ prepareInstr_ self
m_ prepareInstr_ thirdParty
initializeInstr_ self (Cost: 5)
calibrateInstr_ self (Cost: 5)
initializeInstr_ thirdParty (Cost: 2)
calibrateInstr_ thirdParty (Cost: 2)

Figure 2.2: A diagram, depicting the TDG for our sample domain *satellite_lessSimple*. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks.

the initial one, and an evolution encompasses exactly one decomposition by one method. Hence, the planner chose a method for every evolution, on the way to the solution, the moment the solution is committed.

To obtain a domain with choice, we extend our sample domain *satellite_simple* to create the more complex domain *satellite_lessSimple*, which has task costs associated. *satellite_lessSimple* is only illustrated by the TDG in Figure 2.2, preconditions and effects are the same as in *satellite_simple*.

Sample domain *satellite_lessSimple* presents a choice, between *m_loadInstr_0* and *m_loadInstr_1*, for compound task *loadInstr*. Decomposing with method *m_loadInstr_1* evolves into a partial plan that equates to getting an instrument by buying it from a third party and letting this third party prepare the instrument. For this, the task *prepareInstr_thirdParty* is only decomposable into third party related primitive tasks, with the method *m_prepareInstr_thirdParty*. Effectively, buying from the third party forces the tasks *initializeInstr_thirdParty* and *calibrateInstr_thirdParty* into a solution. Deciding to rely on the own stock, with method *m_loadInstr_0* works exactly the other way around.

One can observe, that choosing the third party solution will yield lower cost than choosing the own stock solution. The sum of costs for the own stock solution: (*getInstr_ownStock*, *initializeInstr_self*, *calibrateInstr_self*) is 16 and thus higher than the cost of 14 for the third party solution: (*getInstr_thirdParty*, *initializeInstr_thirdParty*, *calibrateInstr_thirdParty*). However, a planner does not know that. After evolving loadInstr there are two partial plans in the plan space, namely (*getInstr_ownStock*, *perpareInstr_self*) and (*getInstr_thirdParty*, *perpareInstr_thirdParty*) Now evolving (*getInstr_ownStock*, *perpareInstr_self*) would commit to a suboptimal solution, as this would evolve into a solution and the first solution found is returned. This shows the importance of choosing the most promising task network to evolve next in every iteration of plan-based planning, since the wrong choice could lead to a suboptimal plan. The most promising network being the one that eventually can evolve into the most cost-efficient solution. To choose the most promising task network to evolve, the planner needs a way to rank task networks, based on the costs of plans they potentially can evolve into in the future. For this, a *heuristic* is commonly employed in plan-based HTN planning to *inform* or *guide* the planner.

The authors of Bercher, Behnke, Höller, and Biundo [BBHB17] propose a cost aware heuristic for plan-based HTN planning, called TDG_c. TDG_c is based on the TDG and calculates TDG_c cost estimates for methods and compound tasks, if task costs are attached to primitive tasks. We denote TDG_c cost estimates with h_t for tasks and h_m for methods. TDG_c can either be only precalculated or also recalculated in between planning iterations, however we only work with the precalculated version. The manner in which TDG_c calculates cost estimates is conveniently explained when considering our example domain in Figure 2.2. Primitive tasks have a TDG_c cost estimate equal to their cost. The TDG_c cost estimate for a method is calculated by summing up the TDG_c cost estimates for the method's subtasks. For instance, *m_prepareInstr_thirdParty* has a TDG_c cost estimate of four, while *m_prepareInstr_self* has ten. For compound tasks, the TDG_c cost estimate is the minimum cost of all methods targeting this task, so *prepareInstr_thirdParty* has a cost estimate of four and *prepareInstr_self* has a cost estimate of ten. This calculation is repeated until all methods and tasks have TDG_c cost estimates. The cost estimate calculation is formally depicted in Definition 2.3.1 and is adapted from [BBHB17].

With these TDG_c cost estimates, a TDG_c heuristic value can be calculated for a task network, which can inform a plan-based planner of how promising a partial plan is. The TDG_c heuristic value of a task network, is the sum of all TDG_c cost estimates of its tasks, formally defined in Definition 2.3.2. Now, with the TDG_c heuristic value informing a planner, it knows to evolve the partial plan: *getInstr_thirdParty*, *perpareInstr_thirdParty* first, as it has a cost estimate of 14, while *getInstr_ownStock*, *perpareInstr_self* has a cost estimate of 16.

Definition 2.3.1 (TDG_c cost estimate)
Let $G = (V_T, V_M, E_{T\to M}, E_{M\to T})$ be a TDG.
For a task vertex, we set:

$$h_t(v_t) := \begin{cases} \textit{task cost of } v_t & \textit{if } v_t \textit{ is a vertex of a primtive task} \\ \min_{(v_t,v_m)\in E_{T\to M}} h_m(v_m) & \textit{else} \end{cases}$$

For a method vertex, we set:

$$h_m(v_m) := \sum_{(v_m,v_t)\in E_{M\to T}} h_t(v_t)$$

Definition 2.3.2 (TDG_c heuristic value)
We define the heuristic value of a task network Tn as:

$$h_{tn}(Tn) := \sum_{v_t\in Tn} (h_t(v_t))$$

3 Risk-aware Plan-based HTN Planning

After discussing the fundamentals of plan-based HTN planning in the previous chapter, we move on to describing the approach to risk-aware plan-based HTN planning. For this, we introduce risk attitudes and how they can be described with utility functions. Afterwards, we show how to use utility functions to achieve risk-aware plan-based HTN planning. This approach is proposed by Alnazer et al. [AGA22b], therefore, the theoretical fundamentals discussed here are based on their work.

Risk can only be a factor in planning, if an action has multiple potential outcomes with varying desirability. To achieve this Alnazer et al. [AGA22b] extends the HTN planning framework, by assigning multiple outcomes to primitive tasks, with different costs. We assume costs to denote desirability, with higher cost being less desirable. We additionally assume, that all outcomes of a task have the same effect, they only differ in cost. The TDG in Figure 3.1 depicts the domain *satellite_lessSimpleRiskAware* and is an adaption of Figure 2.2 to exemplify this extension, where primitive tasks now have multiple possible costs. For this example domain, the third party is known for sometimes taking a long time to finish their work. This is reflected by *calibrateInstr_thirdParty* and *initializeInstr_thirdParty* having a probability distribution of $p(2) = 0.7$, $p(15) = 0.3$, which represents a 30% chance of the third party taking longer than desired and thus costing more time. We denote one of the possible costs associated with a primitive task tp as $c_i(tp)$, such that $c_0(\text{calibrateInstr\_thirdParty}) = 2$ and $c_1(\text{calibrateInstr\_thirdParty}) = 15$. In contrast to the third party, the own mechanics are known to be consistent, equally reflected in the probability distributions of primitive tasks related to them. One can observe, that choosing the third party solution is risk bearing. However, we are yet to make a planner able to decide on a solution, based on that fact.

3.1 Risk attitudes

A risk attitude represents the willingness of an agent to take risks. Alnazer et al. [AGA22b] defines three risk attitudes, namely risk averse, risk neutral and risk seeking. Risk neutral represents an indifferent stance towards risk. As the names of the other attitudes suggest, an agent with a risk averse attitude avoids risk, while a risk seeking agent is inclined to take risks. An agent with a risk seeking attitude will therefore prefer probability distributions over possible costs, that provide a chance for low costs, even if there is the risk for high costs. In contrast, a risk averse agent prefers probability distributions with low differences between costs. For that reason, a risk attitude implies a subjective cost for an agent, where a risk seeking agent attaches lower subjective costs to high objective costs than the risk averse agent. To model this behaviour of subjective cost calculation, we employ utility functions.

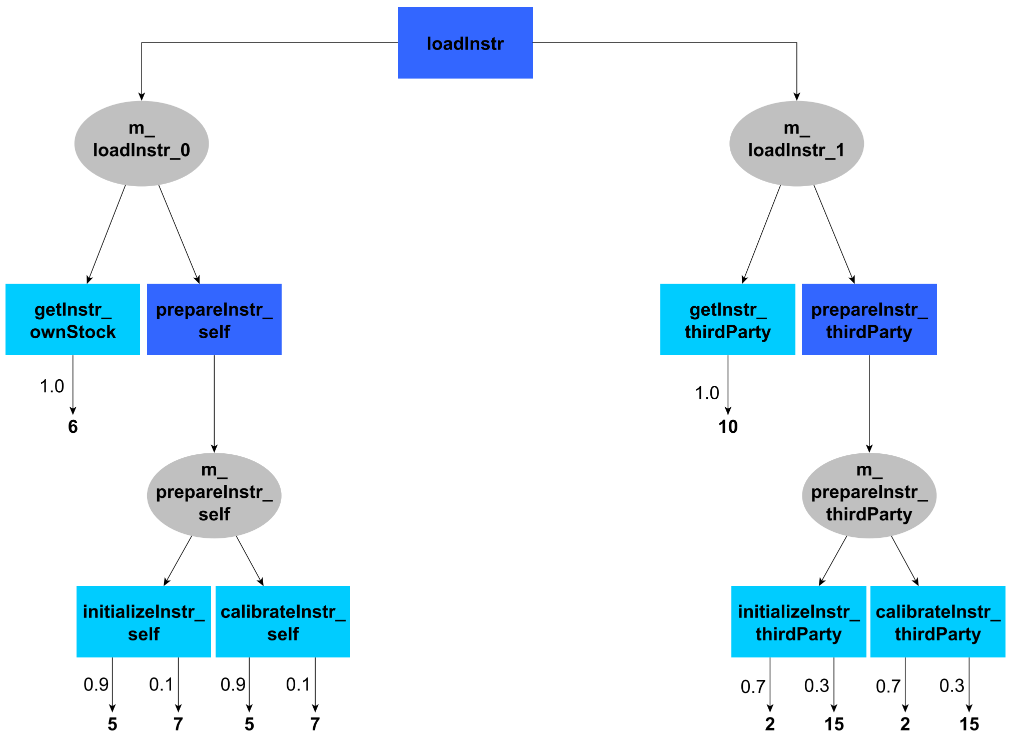


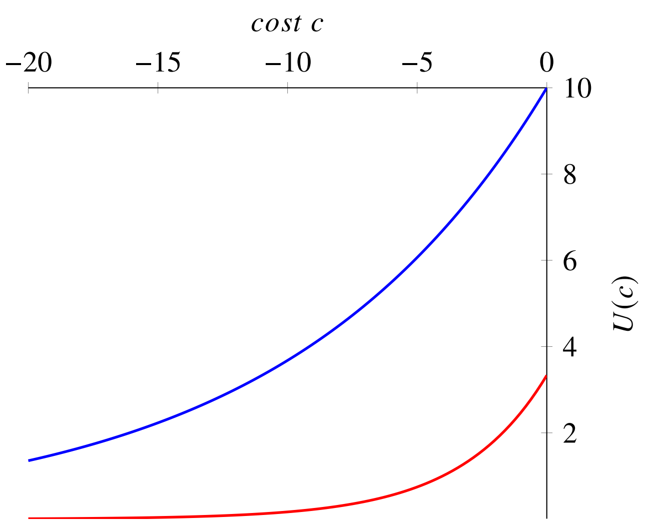
Figure 3.1: A diagram, depicting the TDG for our sample domain *satellite_lessSimpleRiskAware*. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks.

Utility functions are commonly used to describe agents' risk attitudes, since they provide a mapping between objective cost and subjective cost [WK15]. The subjective cost these functions provide is the utility of an objective cost. We employ an exponential utility function, shown in Definition 3.1.1. The parameters a and α allow us to employ a wide range of utility functions, depending on the parameter's values. The parameter a denotes the risk attitude and can either have a value of one or minus one, with $a = 1$ representing the risk seeking attitude and $a = -1$ the risk averse attitude. The risk neutral attitude is indifferent towards risk, and therefore we do not need a mapping to subjective costs, to describe this attitude as we can just consider the objective costs. The parameter α is the curving coefficient of the utility function. We also call this the *intensity*.

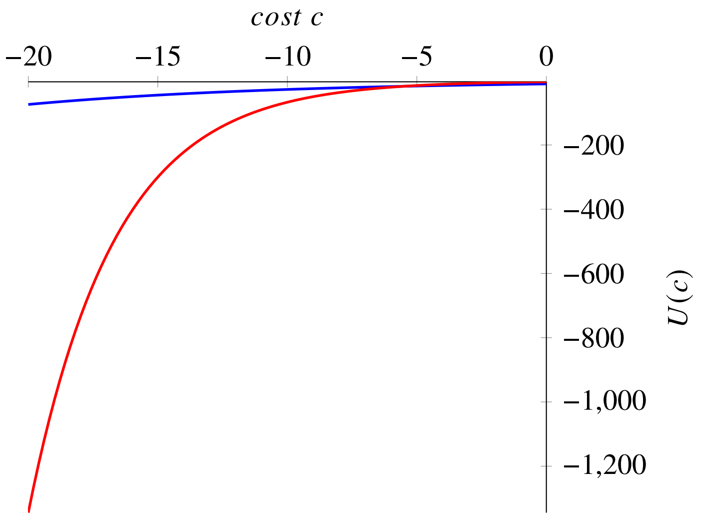
Figure 3.2 illustrates examples of utility functions to describe the behaviour of the utility functions we employ. It should be noted, that we assume cost to be a negative value, for the purpose of utility calculation. As can be seen, the utility for the risk seeking attitude decreases far slower than the utility for the risk averse attitude, making it, so the subjective cost of an objective high cost is lower for risk seeking agents. One can observe, that the intensity impacts the rate at which the utility decreases with cost, which leads to a stronger bias towards solutions, expressing the given risk attitude.

Definition 3.1.1 (Static exponential utility function)
Let $c_i(tp)$ be a possible cost of tp
For the utility, we set:

$$U_c(c_i(tp)) = \begin{cases} c_i(tp) & \textit{if risk attitude is neutral} \\ \frac{a(e^{a\alpha c_i(tp)})}{\alpha} & \textit{else} \end{cases}$$



(a) Utility functions for the risk seeking attitude.



(b) Utility functions for the risk averse attitude.

Figure 3.2: A depiction of the exponential utility functions, employed in this work. The blue function has $\alpha = 0.1$ and the red function has $\alpha = 0.3$.

3.2 Expected Utility

With utility functions, representing risk attitudes, we can define a heuristic that relies on these utility functions to guide planning towards solutions, that adhere to a specified risk attitude. Utilizing this heuristic, together with plan-based planning, achieves risk-aware plan-based planning.

When faced with the decision between multiple probability distributions of costs, it is intuitive to prefer the distribution, which yields the lowest expected cost. In our case, this represents a risk neutral attitude, as this disregards risk. To factor risk consideration into this decision between probability distributions of costs, we can apply a utility function to each cost and so obtain the *expected utility* for each distribution. The decision between probability distributions of costs is then made, according to the risk attitude, the applied utility function represents. Since one probability distribution of costs is attached to one primitive task, this decision equates to choosing a primitive task. We calculate the expected utility of a primitive task tp as defined in Definition 3.2.1. The expected utility for a risk neutral attitude is calculated analogously to how expected cost would be calculated. As we assume costs to be always negative, we multiply the absolute value of the cost with minus one.

Definition 3.2.1 (Expected utility of a primitive task)
for a primitive task tp, we set:

$$\begin{aligned} EU(tp) =& p(c_0(tp)) * U_c(|c_0(tp)| * (-1)) && \textit{if risk attitude is neutral} \\ &+ p(c_1(tp)) * U_c(|c_1(tp)| * (-1)) \\ &+ ... \\ &+ p(c_n(tp)) * U_c(|c_n(tp)| * (-1)) \end{aligned}$$

$$\begin{aligned} EU(tp) =& \log_{10}(|(p(c_0(tp)) * U_c(|c_0(tp)| * (-1)) && \textit{else} \\ &+ p(c_1(tp)) * U_c(|c_1(tp)| * (-1)) \\ &+ ... \\ &+ p(c_n(tp)) * U_c(|c_n(tp)| * (-1)))| * \alpha) * a \end{aligned}$$

With the expected utility, a solution of primitive tasks can be chosen depending on risk attitude. However, a plan-based HTN planner does not know a solution's utility in advance when planning, so, we are in the same position as we were in Section 2.3, and thus need a heuristic. For this, we follow the example of the TDG$_c$ heuristic and calculate *estimated utilities* for compound tasks and methods. The calculation can be described with Definition 3.2.2 and utilizes the TDG just like the heuristic in Definition 2.3.1. Here, we maximize over all methods that decompose a compound task, to obtain the estimated utility of this compound task. This can lead to an infinite calculation if a domain is cyclic, meaning if there is a compound task tc that is decomposed by a method, with subtasks that eventually will be decomposed to tc. The infinite calculation occurs in the maximization over methods and if positive expected utilities are given, as the estimated utility of tc is as sum, calculated over itself and additional subtasks' estimated utilities. For negative expected utilities, cycles are not problematic, since calculating the sum over negative values, with one of the values being the sum itself, will never lead to a higher value. Therefore, as long as costs for probability distributions of primitive tasks are in a value range that leads to negative expected utilities, cycles do not lead to infinite calculation loops.

Definition 3.2.2 (Estimated utility)
Let $G = (V_T, V_M, E_{T\to M}, E_{M\to T})$ be a TDG.
For a task vertex, we set:

$$ESU(v_t) := \begin{cases} EU(v_t) & \textit{if } v_t \textit{ is a vertex of a primtive task} \\ \max_{(v_t,v_m)\in E_{T\to M}} ESU(v_m) & \textit{else} \end{cases}$$

For a method vertex, we set:

$$ESU(v_m) := \sum_{(v_m,v_t)\in E_{M\to T}} ESU(v_t)$$

We are interested in the highest expected utility solution, since this solution expresses the risk attitude most accurately. That is because a utility function is a mapping from objective value to subjective value, and a rational agent will maximize subjective value. To obtain the expected utility of a task network and, therefore, of a solution, we sum up the estimated utilities of all tasks in the task network as described in Definition 3.2.3. With this, we have a heuristic that will never underestimate the cost of a task network. This is the case, as we maximize over methods for compound tasks and calculate a sum for methods. If the A* search algorithm is paired with an admissible heuristic, it is guaranteed to return the lowest possible cost to reach a node. This is also the case for finding the highest possible cost [SKP+14]. A* is related to pathfinding problems, and one might ask how this relates to plan-based HTN planning. This is answered by looking at Algorithm 2.1 and considering task network evolution. One can equate every possible evolution of task networks, to a node in a graph. The edges of the graph represent the methods, evolving the task networks. We utilize pathfinding to find the longest path (highest expected utility path), from the initial task network to a solution task network in this graph. Every node has a heuristic value, which is the expected utility of the task network of that node, and the plan space can be viewed as the open list. Now consider that A* searches for the next node to expand like this: $f(x) = g(x) + h(x)$. In our case, $g(x)$ is the expected utility of the task network currently getting evolved and $h(x)$ is the lowest expected utility of all task networks that can be reached via one decomposition. Therefore, pairing the expected utility heuristic with plan-based HTN planning will always lead to the highest utility solution.

Definition 3.2.3 (Expected utility of a task network)
Let $G = (V_T, V_M, E_{T\to M}, E_{M\to T})$ be a TDG.
For any Tn consisting of tasks, represented by task vertices $(v_{t0}, ..., v_{tn})$, we set:

$$EU_{Tn}(Tn) := ESU(v_{t0}) + ESU(v_{t1}) + ... \; ESU(v_{tn})$$

With this expected utility for a task network, we inform plan-based HTN planning so that it yields the solution, most representative of the chosen risk attitude. To show this, let's consider our example domain in Figure 3.1. Suppose we run planning with an intensity of 0.5 ($\alpha = 0.5$) and with the risk seeking attitude, which means we set $a = 1$. Calculating the expected utility for primitive tasks, yields: (numbers are rounded to six decimal points)

$$\begin{aligned}
EU(\text{initializeInstr\_self}) &= \log_{10}((|0.9 * (\frac{1(e^{1*0.5*-5})}{0.5}) + 0.1 * (\frac{1(e^{1*0.5*-7})}{0.5})|) * 0.5) * 1 \\
&= \log_{10}((|0.147753 + 0.006039|) * 0.5) * 1 \\
&= \log_{10}((0.076896) = -1.114096 = EU(\text{calibrateInstr\_self})
\end{aligned} \tag{3.1}$$

$$\begin{aligned}
EU(\text{initializeInstr\_thirdParty}) &= \log_{10}((|0.7 * (\frac{1(e^{1*0.5*-2})}{0.5}) + 0.3 * (\frac{1(e^{1*0.5*-15})}{0.5})|) * 0.5) * 1 \\
&= \log_{10}((|0.515031 + 0.00033|) * 0.5) * 1 \\
&= \log_{10}(0.257681) = -0.588917 = EU(\text{calibrateInstr\_thirdParty})
\end{aligned} \tag{3.2}$$

$$\begin{aligned}
EU(\text{getInstr\_ownStock}) &= \log_{10}((|1 * (\frac{1(e^{1*0.5*-6})}{0.5})|) * 0.5) * 1 \\
&= \log_{10}((|0.099574|) * 0.5) * 1 \\
&= -1.302884
\end{aligned} \tag{3.3}$$

$$\begin{aligned}
EU(\text{getInstr\_thirdParty}) &= \log_{10}((|1 * (\frac{1(e^{1*0.5*-10})}{0.5})|) * 0.5) * 1 \\
&= \log_{10}((|0.013476|) * 0.5) * 1 \\
&= -2.171469
\end{aligned} \tag{3.4}$$

Now we can calculate the utility estimates for methods:

$$\begin{aligned}
ESU(\text{m\_prepareInstr\_thirdParty}) &= ESU(\text{initializeInstr\_thirdParty}) \\
&+ ESU(\text{calibrateInstr\_thirdParty}) = -0.588917 + -0.588917 = -1.177834
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
ESU(\text{m\_prepareInstr\_self}) &= ESU(\text{initializeInstr\_self}) \\
&+ ESU(\text{calibrateInstr\_self}) = -1.114096 + -1.114096 = -2.228192
\end{aligned} \tag{3.6}$$

Which also tells us the utility estimates for compound tasks *prepareInstr_thirdParty* and *prepareInstr_self*, since they only have one method targeting them and the utility estimate of a compound task is the maximum estimated utility of all methods targeting the compound task. Hence, $ESU(\text{prepareInstr\_thirdParty}) = -1.177834$ and $ESU(\text{m\_prepareInstr\_self}) = -2.228192$.

Having these estimated utilities allows the calculation of the expected utility for task networks. This enables informing the planner of how promising a task network for evolution is, with a task network having a higher utility being more promising. Remember that there are two possible solutions in Figure 3.1, the own stock solution and the third party solution and the third party solution is the more risk laden, as the primitive tasks in the third party solution have a considerable chance for high costs. On top of that, remember that a plan-based planner will evolve the initial task network into two task networks: $Tn_1 = (\textit{getInstr\_ownStock}, \textit{perpareInstr\_self})$ and $Tn_2 = (\textit{getInstr\_thirdParty}, \textit{perpareInstr\_thirdParty})$. The choice of which task network to evolve next is crucial, since the evolution will be either the own stock solution or the third party solution. To inform the planner of which task network is more promising, the expected utility dictates that $EU_{Tn}(Tn_1) = ESU(\text{getInstr\_ownStock}) + ESU(\text{perpareInstr\_self}) = -1.302884 + -2.228192 = -3.531076$ and $EU_{Tn}(Tn_2) = ESU(\text{getInstr\_thirdParty}) + ESU(\text{perpareInstr\_thirdParty}) = -2.171469 + -1.177834 = -3.349303$. Since $-3.349303 > -3.531076$. Expected utility, informs the planner to choose Tn_2 to evolve next, which leads to the solution: (*getInstr_thirdParty*, *initializeInstr_thirdParty*, *calibrateInstr_thirdParty*). This third party related solution is the more risk bearing one, and thus the expected utility informed the plan-based HTN planner in a way that resulted in the solution that expresses a risk seeking attitude. With that, we achieved risk-aware plan-based HTN planning.

The risk neutral attitude represents the absence of subjective risk consideration. Therefore, expected utility for a task network is calculated as the sum of expected costs of tasks. Effectively, the expected utility of a task network for the risk neutral attitude is the expected cost of the task network. With that, risk-aware planning with the risk neutral attitude, informs a planner so that planning leads to the solution with the lowest expected cost and therefore to the expected cost optimal solution. This expected cost optimal solution can be argued to be the objectively optimal solution, as it considers the objective costs, without mapping to subjective costs. In contrast, risk-aware planning with the other two risk attitudes, leads to a solution that deviates from the expected cost optimal solution. The planner accepts an amount of additional expected cost to adhere to the given risk attitude. However, there is a maximum of additional expected cost a risk-aware planner is willing to accept. This maximum is increased and decreased with the increasing and decreasing of the intensity α. With this in mind, we can validate the correctness of this approach of risk-aware HTN planning, by observing if different risk attitudes lead to different solutions, and by investigating the difference in expected cost of these solutions.

4 Implementation

RAPANDA3 is a proof of concept for the risk-aware plan-based HTN planning approach, discussed in the last chapter. RAPANDA3 is build on top of the PANDA3[1] planner. For this reason, chapter four contains a detailed look at PANDA3's source code classes relevant to plan-based HTN planning. Additionally, we introduce the changes we made to these classes, as part of our risk aware adaption, to realize risk-aware plan-based HTN planning. On top of that, we show the format of the output one can expect to receive, when planning with RAPANDA3.

4.1 Overview

PANDA3 is a command line tool for AI planning. It is capable of multiple planning approaches, including state-based HTN planning [HBBB21], however we will only cover plan-based HTN planning, since our risk-aware adaption is based on the plan-based approach. PANDA3 is mainly written in Java and Scala and developed first and foremost by Gregor Behnke, Pascal Bercher and Daniel Höller [PANDA23]. Additional contributions were made on GitHub by a user with the username "ziggystar" and a user named "Kristof Mickeleit". The last changes were committed two years ago and since then, PANDA3 was abandoned.

In PANDA3, the planning with plan-based HTN planning can be roughly divided into three phases, *parsing phase*, *precomputation phase* and *search phase*. The parsing phase marks the start of planning, where a problem instance file and a domain file are committed to PANDA3 in a valid format, such as HDDL. Parsing is conducted with the Another Tool for Language Recognition (ANTLR) runtime version 4.x, where a lexer and parser translate the problem instance and domain files into data structures, that allow the rest of PANDA3's classes access to the information in the domain and problem instance. In the following phase, the precomputation phase, all lifted tasks methods and predicates in the domain are instantiated. This process is also called *grounding*. The internal domain representation, containing these instances, is then pruned with the usage of a reachability analysis. Lastly, in the search phase, a heuristic computes estimates for methods and tasks, in our case estimated utilities based on a TDG, as discussed in Section 3.1. Afterwards, an algorithm similar to Algorithm 2.1, evolves task networks, starting from the initial one defined in the problem instance input file, until a solution is found.

[1] The source code of the PANDA3 version we are using is found here: https://github.com/galvusdamor/panda3core

Listing 4.1 An example for a primitive task with our newly conceived costdist section. This costdist section defines a probability distribution with a 30% chance of *getInstr* costing ten units and a 70% chance of costing 20 units.

```
;--------------- a primitive task with new costdist section ----------------------
(:action getInstr
  :parameters (?i - instrument ?s - satellite)
  :precondition (not (onBoard ?i ?s ))
  :effect (onBoard ?i ?s)
  :costdist (or (0.3(10)) (0.7(20)))   ;<-------- new in RAPANDA3
  )
```

4.2 Parsing Phase

In the following paragraph, we elaborate on the interactions between classes in the parsing phase, which provides insight on how PANDA3 starts its planning process. Mentioned classes are illustrated in the class diagram Figure 4.1, which provides a graphical overview for the following explanations. Before executing PANDA3 for the first time, it is required to auto generate the classes *antlrHDDLLexer.java* and *antlrHDDLParser.java* from *antlrHDDL.g4*, with the ANTLR 4 runtime. The main entry point of PANDA3 is the *main* method in the Scala object *Main*. *Main* has the case class *RunConfiguration* as a subclass, and the *main* method instantiates *RunConfiguration*, which also instantiates *PlanningConfiguration.scala*, since an instance of *PlanningConfiguration.scala* is part of the constructor of *RunConfiguration*. The *main* method calls the method *RunConfiguration.processCommandLineArguments()*, with the command line string, containing command line arguments, as parameter, and the resulting *planning parameters*[2] are saved in the associated instance of *PlanningConfiguration.scala*. Examples for planning parameters are: location of the input files, the format of the input files and maximum allowed execution time. We assume the input domain and problem instance files to be in HDDL format, for which parsing begins after the configuration has been processed. To this end, the file *antlrHDDL.g4* contains grammatical expressions, that allow *antlrHDDLLexer.java* and *antlrHDDLParser.java* to parse HDDL files. In particular, the class *antlrHDDLParser.java* contains subclasses like *antlrHDDLParser.Task_defContext*, to allow access to the information of input HDDL files. With multiple of these subclasses instantiated, the class *hddlPanda3Visitor.java* enables its instances to extract all information, related to the input files, from the parser. The class *hddlPanda3Visitor.java* is instantiated in *parseDomainAndProblem()*, a method of *HDDLParser.scala*, which passes the file content of the input domain and input problem instance to the *hddlPanda3Visitor.java* instance, for information extraction. The actual calling of *parseDomainAndProblem()* is run by *PlanningConfiguration.scala*, in the method *runParsing()*. The case class *PlanningConfiguration.scala* knows what kind of parser to instantiate, thanks to the information encapsulated in the case class *ParsingConfiguration.scala*. With *runParsing()*, the *parsedDomainAndProblem* tuple is produced, which now can be used in the two following phases, precomputation phase and search phase. Besides calling *parseDomainAndProblem()*, the

[2]We call them like so, this is not a term in the PANDA3 source code.

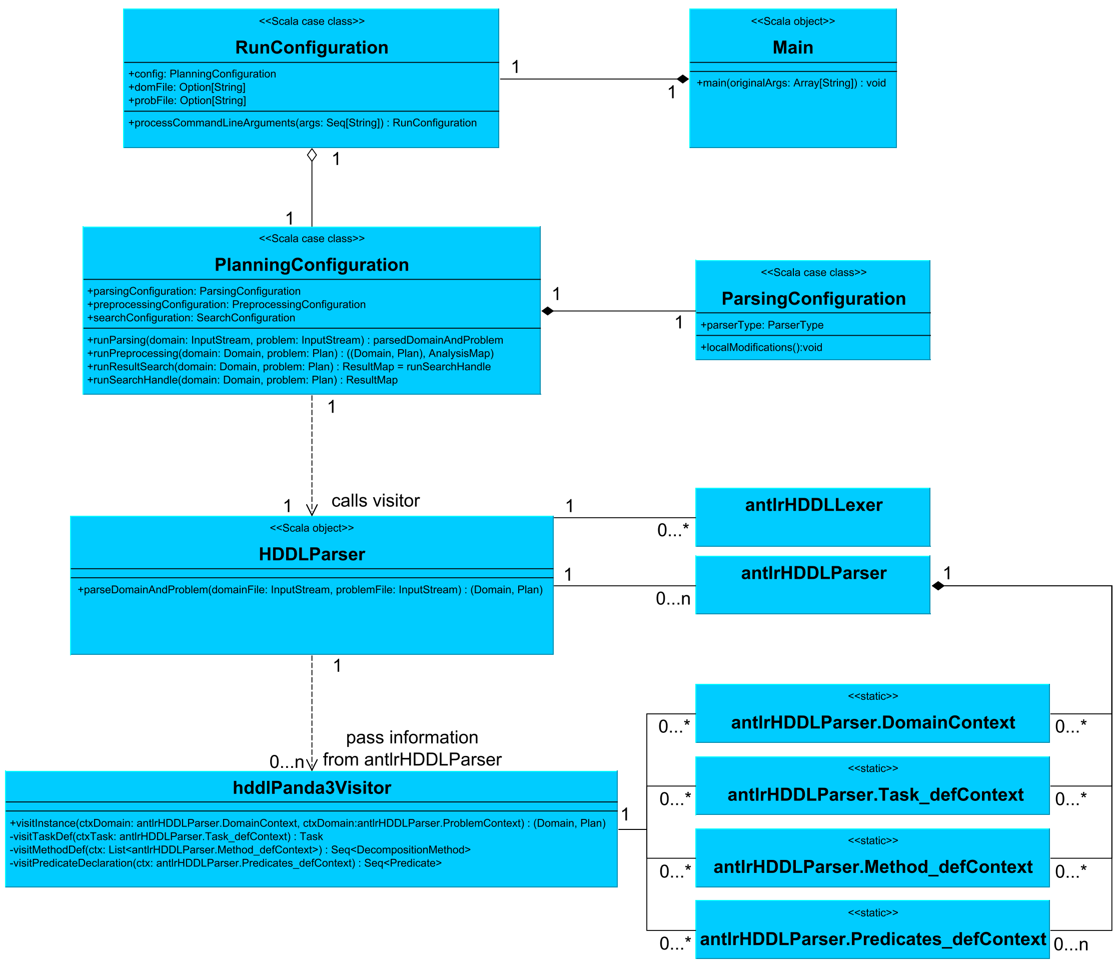

<<Scala case class>>
RunConfiguration
+config: PlanningConfiguration
+domFile: Option[String]
+probFile: Option[String]
+processCommandLineArguments(args: Seq[String]) : RunConfiguration
<<Scala object>>
Main
+main(originalArgs: Array[String]) : void
<<Scala case class>>
PlanningConfiguration
+parsingConfiguration: ParsingConfiguration
+preprocessingConfiguration: PreprocessingConfiguration
+searchConfiguration: SearchConfiguration
+runParsing(domain: InputStream, problem: InputStream) : parsedDomainAndProblem
+runPreprocessing(domain: Domain, problem: Plan) : ((Domain, Plan), AnalysisMap)
+runResultSearch(domain: Domain, problem: Plan) : ResultMap = runSearchHandle
+runSearchHandle(domain: Domain, problem: Plan) : ResultMap
<<Scala case class>>
ParsingConfiguration
+parserType: ParserType
+localModifications():void
calls visitor
<<Scala object>>
HDDLParser
+parseDomainAndProblem(domainFile: InputStream, problemFile: InputStream) : (Domain, Plan)
antlrHDDLLexer
antlrHDDLParser
pass information
from antlrHDDLParser
hddlPanda3Visitor
+visitInstance(ctxDomain: antlrHDDLParser.DomainContext, ctxDomain:antlrHDDLParser.ProblemContext) : (Domain, Plan)
-visitTaskDef(ctxTask: antlrHDDLParser.Task_defContext) : Task
-visitMethodDef(ctx: List<antlrHDDLParser.Method_defContext>) : Seq<DecompositionMethod>
-visitPredicateDeclaration(ctx: antlrHDDLParser.Predicates_defContext) : Seq<Predicate>
<<static>>
antlrHDDLParser.DomainContext
<<static>>
antlrHDDLParser.Task_defContext
<<static>>
antlrHDDLParser.Method_defContext
<<static>>
antlrHDDLParser.Predicates_defContext
1
0...*
0...n

Figure 4.1: A class diagram of classes in PANDA3, taking part in the parsing phase. Only classes relevant to our proof of concept for risk-aware plan-based HTN planning are depicted. All classes, not marked with stereotypes containing “Scala”, are Java classes. All parameters are assumed to be in-parameters, unless marked otherwise. The lists of methods and attributes are abbreviated and therefore not exhaustive, since we would exceed available space otherwise. However, the most important attributes and methods, to understand the relation between classes and the process of parsing in PANDA3, are present.

case class *PlanningConfiguration.scala* is additionally responsible for calling the methods that start the precomputation phase and search phase, therefore this case class is effectively the entry point for these phases.

4.2.1 Modifications

For RAPANDA3, we make the following modifications. The modifications are not reflected in Figure 4.1. We implement several new options for planning parameters (command line arguments) in *PlanningConfiguration.scala*, so that risk-aware plan-based HTN planning can be specified. The new options are:

-riskattitude “averse”|“neutral”|“seeking”
-coefficient *a double type*
-parser “hddlr”
-searchAlgorithm “astar-r”
-heuristic “tdg-r”

The command line argument **-riskattitude** allows setting the risk attitude for planning, while **-coefficient** sets the intensity of our employed utility function. The command line argument **-parser** is already present in PANDA3, and we add the option “hddlr” to enable parsing of risk-aware domains. Likewise, The command line argument **-searchAlgorithm** exists already and has its options expanded with “astar-r”, which specifies the usage of A* to find the highest expected utility. Lastly, the command line argument **-heuristic** receives the “tdg-r” option, that specifies the calculation of utility estimates.

An additional change we implement to this phase is located in *antlrHDDL.g4*, the grammar file for the HDDL input parsing. Here we extend the grammar to make parsing of a new HDDL section possible. The section is called *costdist* and is responsible for representing the cost distribution of a primitive task. The section *:costdist (or (0.3(10)) (0.7(20)))* equates to a probability distribution of $p(10) = 0.3$, $p(20) = 0.7$ for the cost of a primitive task, possessing this section. Even though, for the purpose of utility calculation, we assume cost to be a negative value, the values passed to the costdist section need to be positive, as we think that to be more user-friendly. Cost is measured in abstract units, and what resource is actually consumed is domain dependent. An example usage of costdist is found in Listing 4.1. We call domains, with actions that posses this costdist section, *risk-aware domains*.

Our next modification to the parsing phase is located in the file *hddlPanda3Visitor.java*, where we implement expected utility calculation. The exact location, in which the changes lie, is *visitTaskDef()*, the method responsible for parsing tasks of the domain file. In this method, for primitive tasks, the costdist section is interpreted as a probability distribution and computed into the expected utility, as described in Section 3.1. Afterwards, this expected utility is added to a hash map, with a reference to the primitive task it belongs to. This hash map is used later in the search phase, to read the expected utility for a primitive task.

4.3 Preprocessing Phase

We did not need to make modifications to classes in the preprocessing phase, nonetheless the procedures in this phase are relevant to our risk-aware adaption. The preprocessing phase consists of *grounding* the domain and reachability analysis. Usually, planning requires a grounded representation of the domain [BHBB19]. A grounded or ground domain is present, if all lifted methods, tasks and predicates are instantiated. As can be seen in the example domain 2.1, domains are commonly described in a lifted fashion for a more compact representation. Grounding is the process called to *ground* a domain. Grounding can naively be performed by instantiating all lifted methods, tasks and predicates with all possible combinations of constant bindings for parameter variables. This leads to a sizeable grounded representation of the domain, which usually includes many method, task and predicate instances that will never be relevant in planning for the current problem instance. To prune this grounded representation, reachability analysis is conducted. PANDA3 performs reachability analysis by creating a grounded TDG, with the root vertex being the initial task network of the current problem instance. With this TDG, not reachable instances can be identified and pruned. In addition to the just described one, PANDA3 employs several more preprocessing techniques. These are elaborated on by Behnke et al. [BHBB19]. These techniques induce multiple stages of reachability analysis, including stages for the lifted domain.

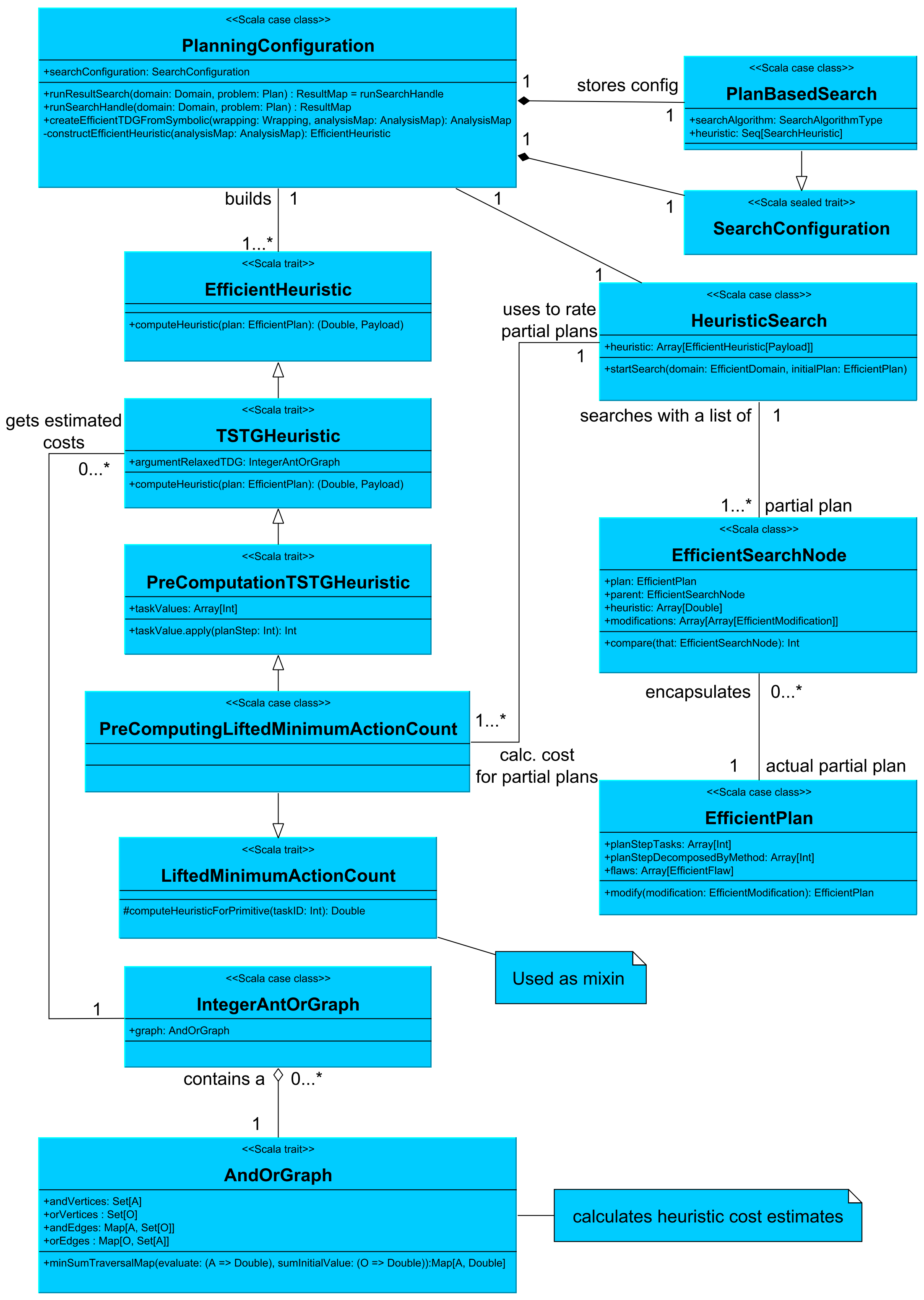
4.4 Search Phase

In the search phase, heuristic estimates are calculated, and a solution is computed. We already discussed in Section 4.2 how the *Main* Scala object instantiates an instance of *PlanningConfiguration.scala*, which calls all three phases. Therefore, in Figure 4.2, that illustrates the relations of classes in the search phase, we do not include the main entry point of PANDA3. For the following explanations, we assume that the TDG_c heuristic is specified in the planning parameters, so that TDG_c is informing the planning. We additionally assume that only one heuristic is informing the planning.

The instance of *PlanningConfiguration.scala* creates an *PreComputingLiftedMinimumActionCount.scala* instance, a class that has all the methods and attributes necessary to heuristically inform planning. That is, because it inherits from multiple Scala traits, providing these methods and attributes. One of the Scala traits from which *PreComputingLiftedMinimumActionCount.scala* inherits is *PreComputationTSTGHeuristic.scala*, which in turn inherits from *TSTGHeuristic.scala*. *TSTGHeuristic.scala* has an instance of *IntegerAntOrGraph.scala* as *argumentRelaxedTDG* attribute, which is a data structure, representing the heuristically estimated costs for methods and tasks. Since we assume TDG_c as heuristic, these estimated costs are calculated as described in Definition 2.3.1. The moment the instance of *PlanningConfiguration.scala* instantiates *PreComputingLiftedMinimumActionCount.scala*, the method *minSumTraversalMap()* in *AndOrGraph.scala*, calculates the cost estimates. The instance of *PlanningConfiguration.scala* creates an instance of *HeuristicSearch.scala* and hands the instance of *PreComputingLiftedMinimumActionCount.scala* to the *HeuristicSearch.scala* instance, where it is stored in the *heuristic* attribute array at the first index. In the same manner as just described, *PlanningConfiguration.scala* can feasibly create multiple instances of *EfficientHeuristic.scala*, that inform the planning with a heuristic. This is why the *heuristic* attribute of *HeuristicSearch.scala* is an array.

The instance of *HeuristicSearch.scala* calls its method *startSearch()*, to start the A* search. *StartSearch()* uses the Scala native data structure *PriorityQueue*, which holds all task networks yet to be evolved. Therefore, *PriorityQueue* represent the plan space in PANDA3 for plan-based HTN planning. These partial plans are represented by instances of *EfficientSearchNode.scala*, which encapsulate instances of *EfficientPlan.scala*, with instances of *EfficientPlan.scala* being the actual partial plans. The way *HeuristicSearch.scala* implements plan-based HTN planning in cooperation with *EfficientSearchNode.scala* and *PreComputingLiftedMinimumActionCount.scala* is depicted in Algorithm 4.1. This algorithm is similar to Algorithm 2.1. However, in Algorithm 4.1, partial plans have flaws to be resolved instead of compound tasks to be decomposed. Since PANDA3 implements more Automated Planning approaches than only plan-based HTN planning, it has to provide different manners, with which to evolve a plan. Since plan-based HTN planning evolves plans by decomposition, a flaw equates to a compound task. The manner in which a flaw to resolve is chosen, is defined by the planning parameters. We assume the flaw choosing algorithm to be Least-Cost Flaw-Repair (LCFR), which chooses the compound task that results in the least amount of evolutions.

After decomposition, the new partial plan is encapsulated in an *EfficientSearchNode* instance. The *newPlanHeuristicValue*, which is computed by *heuristic.computeHeuristic()*, is stored in the *heuristic* attribute of this new *EfficientSearchNode* instance. Since it is feasible that an array of heuristics is informing the planning, the *heuristic* attribute of *EfficientSearchNode* is an array as well. We assume there to be a single heuristic, hence, *newPlanHeuristicValue* is stored at the first



<<Scala case class>>
PlanningConfiguration
+searchConfiguration: SearchConfiguration
+runResultSearch(domain: Domain, problem: Plan) : ResultMap = runSearchHandle
+runSearchHandle(domain: Domain, problem: Plan) : ResultMap
+createEfficientTDGFromSymbolic(wrapping: Wrapping, analysisMap: AnalysisMap): AnalysisMap
-constructEfficientHeuristic(analysisMap: AnalysisMap): EfficientHeuristic
stores config
<<Scala case class>>
PlanBasedSearch
+searchAlgorithm: SearchAlgorithmType
+heuristic: Seq[SearchHeuristic]
<<Scala sealed trait>>
SearchConfiguration
builds
1...*
<<Scala trait>>
EfficientHeuristic
+computeHeuristic(plan: EfficientPlan): (Double, Payload)
uses to rate partial plans
<<Scala case class>>
HeuristicSearch
+heuristic: Array[EfficientHeuristic[Payload]]
+startSearch(domain: EfficientDomain, initialPlan: EfficientPlan)
gets estimated costs
0...*
<<Scala trait>>
TSTGHeuristic
+argumentRelaxedTDG: IntegerAntOrGraph
+computeHeuristic(plan: EfficientPlan): (Double, Payload)
searches with a list of
1...*
partial plan
<<Scala class>>
EfficientSearchNode
+plan: EfficientPlan
+parent: EfficientSearchNode
+heuristic: Array[Double]
+modifications: Array[Array[EfficientModification]]
+compare(that: EfficientSearchNode): Int
<<Scala trait>>
PreComputationTSTGHeuristic
+taskValues: Array[Int]
+taskValue.apply(planStep: Int): Int
<<Scala case class>>
PreComputingLiftedMinimumActionCount
1...*
calc. cost for partial plans
encapsulates
0...*
actual partial plan
<<Scala case class>>
EfficientPlan
+planStepTasks: Array[Int]
+planStepDecomposedByMethod: Array[Int]
+flaws: Array[EfficientFlaw]
+modify(modification: EfficientModification): EfficientPlan
<<Scala trait>>
LiftedMinimumActionCount
#computeHeuristicForPrimitive(taskID: Int): Double
Used as mixin
<<Scala case class>>
IntegerAntOrGraph
+graph: AndOrGraph
contains a
0...*
<<Scala trait>>
AndOrGraph
+andVertices: Set[A]
+orVertices : Set[O]
+andEdges: Map[A, Set[O]]
+orEdges : Map[O, Set[A]]
+minSumTraversalMap(evaluate: (A => Double), sumInitialValue: (O => Double)):Map[A, Double]
calculates heuristic cost estimates

Figure 4.2: A class diagram of classes in PANDA3, taking part in the search phase. Only classes relevant to our proof of concept for risk-aware plan-based HTN planning are depicted. All classes, not marked with stereotypes containing “Scala”, are Java classes. All parameters are assumed to be in-parameters, unless marked otherwise. The lists of methods and attributes are abbreviated and therefore not exhaustive, since we would exceed available space otherwise. However, the most important attributes and methods, to understand the relation between classes in the search phase, are present.

index of the *heuristic* attribute array. The method *computeHeuristic()* is called by an instance of *PreComputingLiftedMinimumActionCount.scala* and computes the TDG_c heuristic value, based on the TDG_c cost estimates, as described in Definition 2.3.2. Afterwards, this new *EfficientSearchNode* is added to the *PriorityQueue*, which then reshuffles all instances of *EfficientSearchNode* in itself. This reshuffle is based on an ordering the *compare()* method of *EfficientSearchNode* dictates. The ordering ensures that the *EfficientSearchNode* instance, with the lowest value in the *heuristic* attribute array at the first index, is brought to the front of the *PriorityQueue* and assigned to $partialPlan_{current}$ in the next iteration. This is the TDG_c heuristic, informing the plan-based planning of the most promising task network to evolve next, as discussed in Section 2.3. If multiple values are present in the *heuristic* array, then these are used for tie-braking.

4.4.1 Modifications

Our changes to the search phase, to obtain RAPANDA3, are explained in the following and not depicted in Figure 4.2. The classes that provide *HeuristicSearch.scala* with the possibility to calculate TDG_c heuristic values are: *PreComputingLiftedMinimumActionCount.scala*, *TSTGHeuristic.scala*, *PreComputationTSTGHeuristic.scala* and *LiftedMinimumActionCount.scala*. We follow the example set by these classes and implement classes that mirror their functionality, except they provide the possibility to calculate expected utility for task networks as described in Definition 3.2.3. The classes we implement are: *PreComputingLiftedMaxUtil.scala*, *TSTGRHeuristic.scala*, *PreComputationTSTGRHeuristic.scala* and *LiftedMaxUtil.scala*. To achieve the calculation of expected utility, we have the estimated utilities calculated by a function we implement, called *maxProductTraversalMap()*, located in *IntegerAntOrGraph.scala*, just underneath the method *minSumTraversalMap()*. *MaxProductTraversalMap()* calculates estimated utilities as described in Definition 3.2.2 and stores them in the *argumentRelaxedTDG* attribute in the class *TSTGRHeuristic.scala*.

Another change we undertake is introducing parameters to several classes and methods so that the *compare()* method of *EfficientSearchNode.scala* forces the *EfficientSearchNode.scala* instance with the highest expected utility in front of the *PriorityQueue*. With this, the chosen risk attitude informs the planner of the most promising partial plan to evolve next.

4.4.2 Output

The output of PANDA3, and therefore RAPAND3, is displayed on the command line and amounts to a list of tasks and methods. This list is the solution RAPAND3 suggests. An example is shown in Listing 4.2, where one can observe that the numbered list has two parts, a part above *root 10* and a part below. The part below describes what methods decomposed what compound task to get to the solution. The numbers behind a method name indicate the numbers of the list lines, that resulted from this decomposition. However, we are only interested in the solution, which is the part above *root*. Here, one can observe the plan steps, that comprise the solution. These are mostly instances of primitive tasks, like *reload car0 depot0 reloading0*, however there are plan steps present, prefixed with "__method_precondition". These are pseudo tasks to model method preconditions, thus we ignore them when examining solutions.

Listing 4.2 An example output of RAPANDA3, which shows a solution with primitive tasks above *root 10*.

```
0 do_nothing car0
1 reload car0 depot0 reloading0
2 __method_precondition_m_change_depot_2_10_precondition car0
3 exit_depot depot0 car0
4 __method_precondition_m_change_depot_0_8_precondition car0
5 enter_depot depot2 car0
6 maintain car0 depot2 maintaining0
7 __method_precondition_m_change_depot_2_10_precondition car0
8 exit_depot depot2 car0
9 departure_no_clean car0 a0
root 10
10 do_assignment a0  -> <m_do_assignment;prepare_car[car0];m_prepare_car_1;0;-1,1> 12 11
11 depart car0 a0  -> m_depart_1 9
12 reload_car car0 reloading0  -> m_reload_car_1 14 1 13 15
13 change_depot car0  -> m_change_depot_2 2 3
14 change_depot car0  -> m_change_depot_3 0
15 maintain_car car0 maintaining0  -> m_maintain_car_0 17 6 16
16 change_depot car0  -> m_change_depot_2 7 8
17 change_depot car0  -> m_change_depot_0 4 5
```

Algorithm 4.1 The algorithm of plan-based HTN planning in PANDA3, encapsulated in *HeuristicSearch.scala*.

```
procedure StartSearch(EfficientPlan initial, PreComputingLiftedMinimumActionCount heuristic)
    initialPlanHeuristic ← heuristic.computeHeuristic(initial)
    PriorityQueue ← newEfficientSearchNode(initial, initialPlanHeuristicValue)
    return heuristicSearch()
end procedure
procedure heuristicSearch()
    while PriorityQueue != ∅ do
        $partialPlan_{current}$ ← PriorityQueue.getAndDel(0)
        if hasNoFlaws($partialPlan_{current}$) then
            return $partialPlan_{current}$
        end if
        flawToResolve ← chooseFlaw($partialPlan_{current}$.flaws)    // a compound task is a flaw
        for all modifications m resolving flawToResolve do    // e.g. methods
            $partialPlan_{new}$ ← $partialPlan_{current}$.modify(m)    // decompose
            newPlanHeuristicValue ← heuristic.computeHeuristic($partialPlan_{new}$)
            PriorityQueue.add(newEfficientSearchNode($partialPlan_{new}$, newPlanHeuristicValue))
            PriorityQueue.reshuffle    // based on EfficientSearchNode.compare
        end for
    end while
    return unsolvable
end procedure
```

5 Evaluation

After implementing risk-aware plan-based HTN planning in RAPANDA3, we design several risk-aware domains with according problem instances in HDDL format[1]. By experimenting on RAPANDA3 with these problem instances and risk-aware domains, we examine the implemented approach. Some of the risk-aware domains are based on International Planning Competition (IPC) 2020 benchmark domains or predecessors of those [IPCa]. Since benchmark domains of IPC 2020 are not risk aware and often do not have high freedom of choice when it comes to methods, we had to design risk-aware domains ourselves. For every risk-aware domain that is a derivative, we firstly introduce the original domain, followed by a description of the changes we made to obtain a risk-aware version of the domain. The name of the risk-aware version of the domain is suffixed with "-RA". For our completely self-designed domain, we only provide an introduction. After the description of the changes to a domain, we present the results of our experiments with the risk-aware domain version, elaborating on the influence of risk attitudes to planning.

The experimental settings are as follows: The experiments are conducted with 28 GB RAM maximum Java heap space and with an *AMD Ryzen 7* processor, boasting 8 cores, each having 3600 MHz. For every plan computation, we set a planning time limit of 15 minutes, since most plan computations, taking longer than 15 minutes, overburden the Java heap space such that the computation crashes. Experiments are conducted with a utility function, having an intensity of $\alpha = 0.5$, unless stated otherwise. Intensity is described in Section 3.1.

We run plan computation for the risk-aware domains and problem instances to compare risk attitudes on solution length, expected cost, expected utility and planning time[2]. Seeing as RAPANDA3 is resource intensive, plan computation for many complex problem instances exceed our planning time limit of 15 minutes. Exploring the limit of complexity, induced by that fact, is one of the goals of this evaluation. The information gleamed from an´experiment is specific to the domain it is conducted with and only holds true there, unless stated otherwise.

5.1 Robot Domain

A totally ordered benchmark domain of IPC 2020, depicting one robot picking up and delivering packages from room to room. The robot has a current room it is located in and can traverse to a different room through a door. A door connects exactly two rooms and can either be closed or open.

[1]The domains and problem instances can be requested at the IAAS Service Computing Department; university Stuttgart

[2] Some computations take several minutes, which makes it not feasible to run these ten to 20 times to calculate an average planning time. Consequently, such computations are only run once.

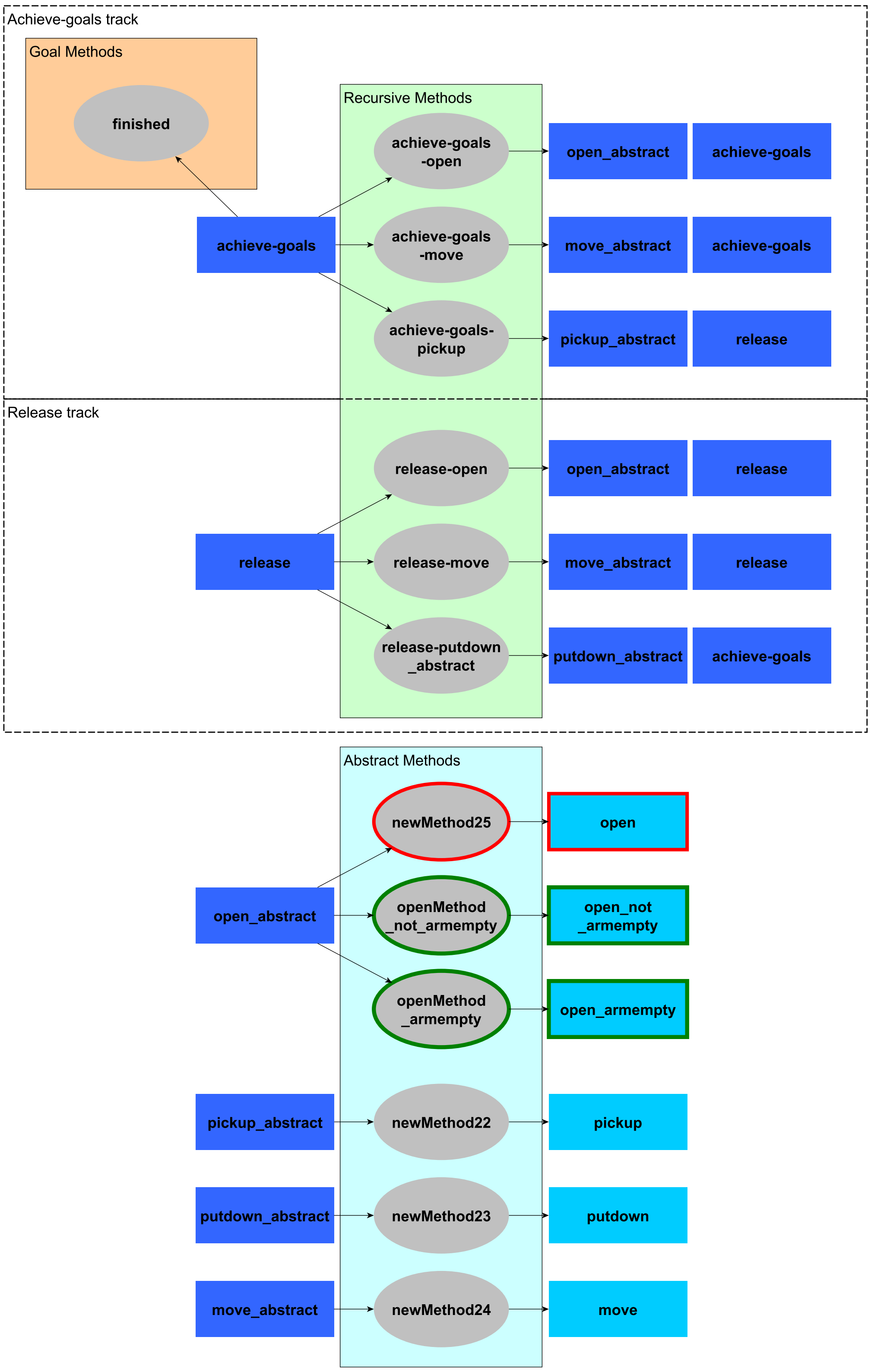


Figure 5.1: A diagram, illustrating the relations of methods and tasks in the *Robot-RA* domain. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks. Green borders indicate additions to the original domain, and red borders indicate removals from the original domain.

Listing 5.1 The available lifted predicates in the *Robot* and *Robot-RA* domain.

```
(:predicates
  (armempty)
  (rloc ?loc - ROOM)
  (in ?obj - PACKAGE ?loc - ROOM)
  (holding ?obj - PACKAGE)
  (closed ?d - ROOMDOOR)
  (door ?loc1 - ROOM ?loc2 - ROOM ?d - ROOMDOOR)
  (goal_in ?obj - PACKAGE ?loc - ROOM)
  )
```

The robot can open a door if the door connects the robot's current room, however it cannot close a door. The robot can pick up or drop a package and can only carry one package at a time. A package has a current room it is located in and a goal room, the robot has to deliver the package to.

The lifted predicates portray packages in a room network by offering the types *PACKAGE*, *ROOM* and *ROOMDOOR*, which is shown in Listing 5.1. Since there is only one robot, it is implicitly given without a type. The robot is supposed to carry only one package at a time, ensured by the *armempty* predicate, which needs to be true for the robot to pick up a package. What package is currently carried is stored in the *holding* predicate. The predicate *rloc* represents the current location of the robot. The *ROOMDOOR* type connects two *ROOM* types via the predicate *door* and is either closed or open, depending on the predicate *closed*. A package is initially located in a *ROOM* and has a goal *ROOM* to be delivered to, represented by predicates *in* and *goal_in*.

Figure 5.1 illustrates the relation between tasks and methods in the robot domain. The following paragraph elaborates on this further. For the sake of explanation we divide the decomposition methods into three equivalence classes, recursive methods, abstract methods and goal methods. All recursive methods decompose either the *achieve-goals* or *release* compound task. The subtasks of these methods are always a pair of compound tasks, including *achieve-goals* or *release* as the second task, enabling recursion. The first task of the pair is decomposable by exactly one abstract method. Abstract methods always decompose into exactly one primitive task from a compound task, named after the primitive task. *Achieve-goals* can only be decomposed to *release* with the method *achieve-goals-pickup*, which introduces the compound task *pickup_abstract*, forcing the injection of the *pickup* primitive task into the solution. From *release* to *achieve-goals* works the other way around, with the difference of injecting the *putdown* primitive task. This gives a planner two recursion tracks to follow, the *achieve-goals-track* and the *release-track* Thanks to this recursion, an arbitrary number of compound tasks, only decomposable by abstract methods, can be chained. Since abstract methods all decompose to exactly one primitive task, this is effectively chaining primitive tasks. Goal methods only include the *finished* method, which posses no subtasks, ending the chaining process if applied. Only *achieve-goals* is decomposable by finished, which makes it so the *release-putdown_abstract* method has to be applied at some point, forcing the planner to have the *putdown* task before finishing, thus having an empty arm for the robot.

To achieve risk awareness, we design the *Robot-RA* domain, which makes the following changes to the above described original. A new primitive task is introduced *open_not_armempty* only applicable if the robot's arm is not empty, representing the robot opening a door while carrying a package. The original primitive task for opening doors is changed to *open_armempty* only applicable if the robot has its arm empty. Furthermore, a new method responsible for decomposing into *open_not_armempty* was added, while the equivalent one was analogously changed to decompose into *open_armempty*. With this, a planner has a choice between *open_not_armempty* and *open_armempty* when decomposing *open_abstract*. The intent here is to make a planner consider if it wants to take the safe option of having the robot's arm empty when opening doors, wasting time backtracking afterwards to get a package, or taking the risk of opening with a package in arm. To depict this risk, all primitive tasks receive the costdist section, with a distribution of (1(15)), representing a 100% chance that the robot takes 15 seconds to perform the respective task. The only exception being the risk bearing task *open_not_armempty*, which received a distribution of ((0.99(15)) (0.01(100))), depicting the risk of the robot needing 100 seconds to open a door, due to having its arm full.

5.1.1 Experiment

We designed the HDDL problem instance *pfile_RA01*[3], which places four connected rooms *c, r1, r2, r3*, with one closed door *d23* between *r2* and *r3*, and two packages *o1* and *o2*, *o1* being right where the robot starts in room *c*. The name of a door denotes the rooms it connects. The rooms are connected, so that the whole room network results in one straight corridor, meaning *c* is only connected to *r1*, *r1* is only connected to *r2* and *r2* is only connected to *r3*. The goal of this problem instance is to have the packages' locations be *r3* and *r1* for *o1* and *o2*, respectively.

In Listing 5.2 three solutions to this problem instance are presented, each of them computed with a different risk attitude, which allows for a direct comparison. A difference in solution can be observed in the first step already. While the risk averse planning goes out of its way to open closed door *d23* with an empty arm, shown in step two, the risk neutral and risk seeking planning pick up package *o1* immediately in step zero and open the door the risky way in step four. This results in the risk averse solution being longer and having a higher expected cost, since the robot has to go back for package *o1* in step eight, showing the risk averse attitude avoiding risk even if expected value suggests otherwise.

For further experimentation, we create three more problem instances, each modifying *pfile_RA01* in a distinct way. *pfile_RA01_more_closed_00* modifies *pfile_RA01*, through closing the door between rooms *r1* and *r2*, adding one more closed door the robot encounters. To obtain *pfile_RA01_more_rooms_00*, we add one additional room *r4* to *pfile_RA01*, connected to *r3* with one open door, resulting in a longer corridor of rooms with only one closed door. Furthermore, package *o1* now has to be delivered to the last room *r4*. For the problem *pfile_RA01_more_packages_00*, we modify *pfile_RA01* by adding two more packages *o3 and o4* that need delivery, increasing the amount of back and forth routing for the robot. Another three problem instances are created through modification of the just introduced problems instances. We create *pfile_RA01_more_closed_01*, by removing package *o2* from *pfile_RA01_more_closed_00*, simplifying the problem. With this, the robot still encounters two locked doors, but has to deliver only one package. The problem instance *pfile_RA01_more_rooms_01* is conceived, by opening the one closed door *d23*. For

[3]Our self designed problem instance can be viewed in the Appendix A.1

Listing 5.2 The primitive task sequences of three computed plans by RAPAND3 in the *Robot-RA* domain. All plans are solutions for our self designed problem instance *pfile_RA01* and each solution is computed with a different risk attitude.

```
;; risk averse attitude, expected cost: 180, expected utility: -10^{39.09}
0 move c r1 d01
1 move r1 r2 d12
2 open_armempty r2 r3 d23
3 __method_precondition_achieve-goals-pickup_3_precondition r2 o2
4 pickup o2 r2
5 move r2 r1 d12
6 __method_precondition_release-putdown_abstract_0_precondition r1 o2
7 putdown o2 r1
8 move r1 c d01
9 __method_precondition_achieve-goals-pickup_3_precondition c o1
10 pickup o1 c
11 move c r1 d01
12 move r1 r2 d12
13 move r2 r3 d23
14 __method_precondition_release-putdown_abstract_0_precondition r3 o1
15 putdown o1 r3

;; risk neutral attitude, expected cost: 150.85, expected utility: -150.85
0 __method_precondition_achieve-goals-pickup_3_precondition c o1
1 pickup o1 c
2 move c r1 d01
3 move r1 r2 d12
4 open_not_armempty r2 r3 d23
5 move r2 r3 d23
6 __method_precondition_release-putdown_abstract_0_precondition r3 o1
7 putdown o1 r3
8 move r3 r2 d23
9 __method_precondition_achieve-goals-pickup_3_precondition r2 o2
10 pickup o2 r2
11 move r2 r1 d12
12 __method_precondition_release-putdown_abstract_0_precondition r1 o2
13 putdown o2 r1

;; risk seeking attitude, expected cost: 150.85, expected utility: 10^{-32.58}
0 __method_precondition_achieve-goals-pickup_3_precondition c o1
1 pickup o1 c
2 move c r1 d01
3 move r1 r2 d12
4 open_not_armempty r2 r3 d23
5 move r2 r3 d23
6 __method_precondition_release-putdown_abstract_0_precondition r3 o1
7 putdown o1 r3
8 move r3 r2 d23
9 __method_precondition_achieve-goals-pickup_3_precondition r2 o2
10 pickup o2 r2
11 move r2 r1 d12
12 __method_precondition_release-putdown_abstract_0_precondition r1 o2
13 putdown o2 r1
```

	Risk attitude	SL	Exp. cost	Exp. utility	Time
pfile_RA01	**Risk averse**	12	180.00	$-10^{39.09}$	181.12s
	Risk neutral	10	150.85	-150.85	9.40s
	Risk seeking	10	150.85	$10^{-32.58}$	10.70s
pfile_RA01_more_closed_00	**Risk averse**	-	–	–	-
	Risk neutral	11	166.70	-166.70	55.91s
	Risk seeking	11	166.70	$10^{-35.84}$	57.41s
pfile_RA01_more_closed_01	**Risk averse**	11	165.00	$-10^{35.83}$	103.10s
	Risk neutral	10	150.85	-150.85	21.64s
	Risk seeking	10	150.85	$10^{-32.58}$	19.68s
pfile_RA01_more_rooms_00	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	-	–	–	-
pfile_RA01_more_rooms_01	**Risk averse**	11	165.00	$-10^{35.83}$	91.74s
	Risk neutral	11	165.00	-165.00	92.69s
	Risk seeking	11	165.00	$10^{-35.83}$	96.66s
pfile_RA01_more_rooms_02	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	-	–	–	-
pfile_RA01_more_packages_00	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	-	–	–	-
pfile_RA01_more_packages_01	**Risk averse**	-	–	–	-
	Risk neutral	13	195.85	-195.85	137.24s
	Risk seeking	13	195.85	$10^{-42.35}$	122.91s

Table 5.1: The results of running plan computation for the listed problem instances with RAPANDA3 in the *Robot-RA* domain. A row refers to one problem instance and has three sub-rows associated, each referring to plan computation with the listed risk attitude. Solution length (SL) refers to the amount of plan steps the solution holds, with one plan step being a primitive task. Exp. cost displays the sum of expected costs of the plan steps. Exp. utility depicts the solution's expected utility rounded to two decimal points, and the time column holds the planning time for RAPANDA3 to compute the solution in seconds, rounded to two decimal points. Any computation that exceeded 15 minutes was aborted, and the cells are marked with -, for the problem instance exceeded our planning time limit.

pfile_RA01_more_packages_01 we remove package *o4* from *pfile_RA01_more_packages_00* to simplify the problem. Lastly, we create *pfile_RA01_more_rooms_02* out of *pfile_RA01_more_rooms_01* by adding one more room to the corridor and requiring the robot to deliver to this last room.

The results of our experiments are compiled into Table 5.1, where many computations can be observed that exceeded our planning time limit, indicating that the showcased problem instance *pfile_RA01* was close to the limit of problem instance complexity we want to explore. Where this limit lies, is indicated in detail by the following observations. Adding one more room is only computable if we simplify, for instance by opening all doors. Adding two rooms oversteps the limit, regardless of opened or closed doors. At maximum, one more package can be included, although this results in a planning time, exceeding 15 minutes, for risk aversion. Similarly, for one additional closed door, risk aversion exceeds our planning time limit, while other attitudes do not. Only after simplifying the problem instance *pfile_RA01_more_closed_00*, by removing one package, does risk aversion generate a solution in time. Planning with risk aversion exceeding 15 minutes is an observable trend in line with solution length tending to be larger for risk averse solutions. On top of that, risk aversion results in a significantly longer planning time, if it does not exceed 15 minutes. This is likely due to the risk laden primitive task *open_armempty* constituting a shortcut for solution length and risk aversion staying away from this risk. Risk neutrality computes the same solutions as the risk seeking attitude, which makes sense if one keeps in mind, that this domain offers effectively two choices for planning: take the risky approach, opening doors with a full arm, or the safe approach. There is no middle ground for risk neutrality to take, so it sides with one of the other attitudes. With which attitude it sides depends on the expected cost of the distribution induced by the *costdist* section for the risky primitive task *open_not_armempty* and the amount of plan steps the risk seeking solution saves compared to the risk avoiding one. Every plan step, except *open_not_armempty*, costs 15 seconds, meaning *open_armempty* must have an expected cost lower than 15 multiplied with the amount of plan steps saved by using *open_not_armempty* for risk neutrality to side with risk aversion. This however seems to only be the case for the problem instance *pfile_RA01_more_rooms_01*, where solution lengths do not differ. In fact, all the solutions seem to be the same for this problem instance. Since for all the other problem instances, where solution length differs, risk aversion leads to a higher expected cost, it can be assumed that risk neutrality will always side with the risk seeking attitude. So the question becomes how much more expected cost is risk averse computation willing to take to stay safe. This not only depends on the cost values of the *costdist* section but also on the intensity of the utility function.

5.2 Satellite Domain

This totally ordered domain is not featured in the [IPCa] and taken from [BBHB17]. It depicts satellites, adjusting directions and switching through instruments to take images of celestial phenomenons. To take an image of a celestial phenomenon, a satellite must point to the direction the phenomenon is in. Additionally, the satellite, must have an instrument on board that supports the mode, with which the image has to be taken. Lastly, the instrument needs to be switched on and calibrated before taking the image. Calibration is done after pointing the satellite in the calibration direction, as dictated by the instrument.

The predicates are listed in 5.3, showing six types, *direction*, *calib_direction* an extension of direction, *image_direction* an extension of direction, *instrument*, *mode* and *satellite*. We made additions to obtain *Satellite-RA*, which are already shown in the listing and marked accordingly, the original *Satellite* domain does not have these predicates.

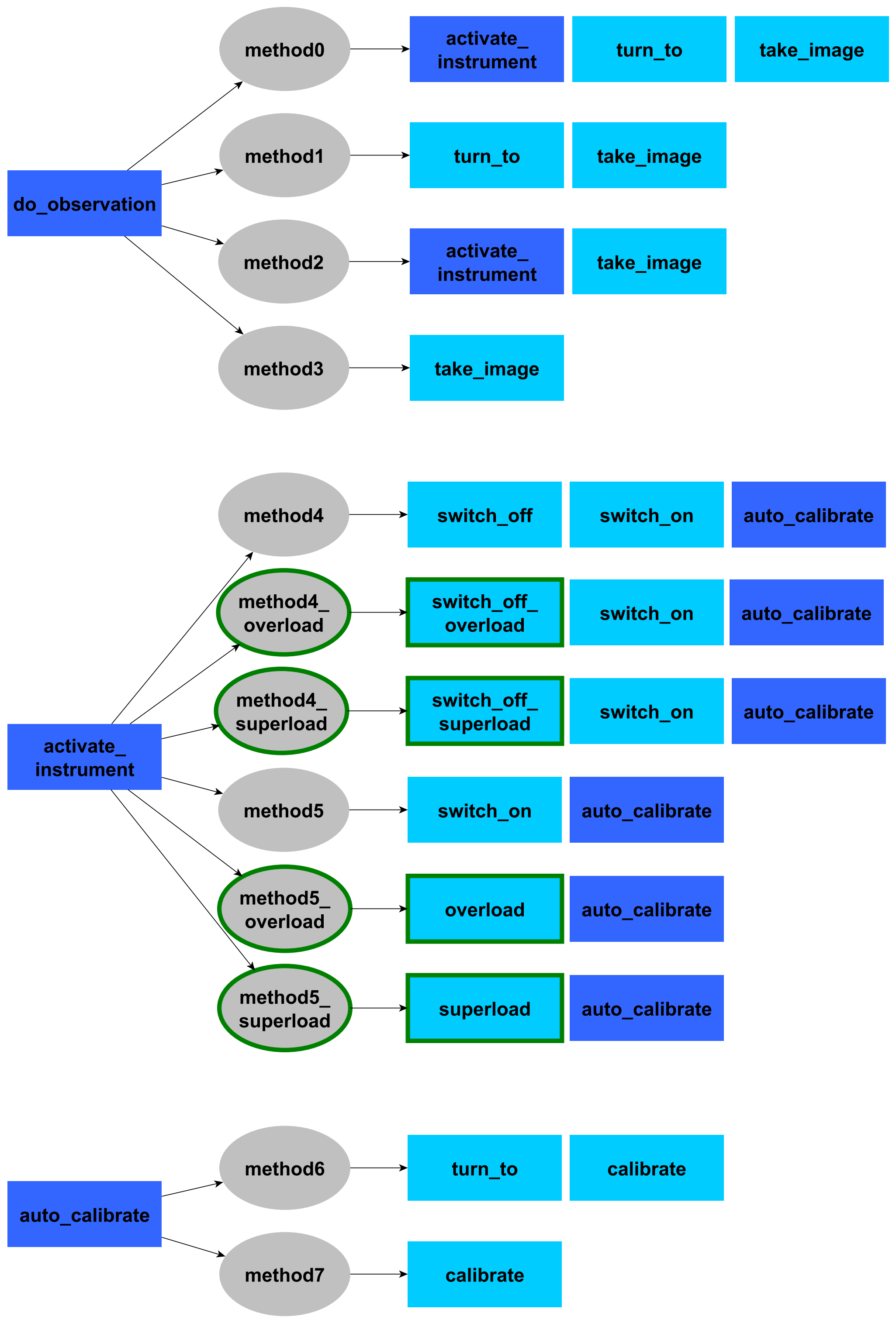


Figure 5.2: A diagram, illustrating the relations of methods and tasks in the *Satellite-RA* domain. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks. Green borders indicate additions to *Satellite-RA*.

Listing 5.3 The available lifted predicates in the *Satellite-RA* domain.

```
(:predicates
  (calibrated ?arg0 - instrument)
  (calibration_target ?arg0 - instrument ?arg1 - calib_direction)
  (have_image ?arg0 - image_direction ?arg1 - mode)
  (on_board ?arg0 - instrument ?arg1 - satellite)
  (pointing ?arg0 - satellite ?arg1 - direction)
  (power_avail ?arg0 - satellite)
  (power_on ?arg0 - instrument)
  (supports ?arg0 - instrument ?arg1 - mode)

      ;; risk-aware addition below
  (overloads ?arg0 - instrument ?arg1 - satellite)
  (overloaded ?arg1 - satellite)
  (superloads ?arg0 - instrument ?arg1 - satellite)
  (superloaded ?arg1 - satellite)
)
```

Multiple *do_observation* compound tasks usually comprises the initial task network of problem instances, associated with this domain. *Do_observation* has two parameter variables of the types *image_direction* and *mode*, *mode* being, for instance, infrared. To do an observation, a satellite needs to be pointing to the *image_direction*. In what direction a satellite is currently pointing is represented by *pointing*. Additionally, an *instrument* supporting the required mode needs to be on board, powered and calibrated. An instrument can support modes through the predicate *supports* and is on board if the predicate *on_board* holds true for a satellite and instrument. Instruments are powered with the predicate *power_on*, however only one power source per satellite is available, denoted by *power_avail*. The predicate *calibrated* depicts if an instrument is calibrated. For calibration, an instrument has a *calib_direction* represented by *calibration_target*. A satellite has to point to this direction before calibrating the instrument.

Figure 5.2 depicts the hierarchy present in the *Satellite-RA* domain. The highest order compound task is *do_observation*, passing along an *image_direction* and *mode* for the image that has to be taken. The planner has four methods to decompose *do_observation*, all of which are alternatives to each other. The adequate choice depends on the current state, as every method decomposes to tasks that might need to be accomplished first before a picture can be taken. For example, *method0* decomposes into tasks that will eventually lead to turning the satellite and activating and calibrating the appropriate instrument, while *method3* leads to only the primitive task of *take_image*, skipping all preliminary tasks. A satellite can have multiple instruments on board, but only one can be powered at once. The compound task *activate_instrument* is decomposable into tasks that deal with this restriction, namely the primitive tasks: *switch_off* and *switch_on*, allowing the redistribution of power. Every time an instrument is switched back on, it needs to be calibrated again, which is depicted by the *auto_calibrate* compound task, that decomposes into either the primitive task: *calibrate*, or the pair of primitive tasks: (*turn_to*,*calibrate*) and the pair might be necessary, if the solution requires turning to the *calibration_target* first.

Our changes to obtain *Satellite-RA*, enables satellites to power additional instruments, under the risk of a power failure, after which, time is wasted as the satellite has to be made operational again. To this end, we introduce two primitive tasks *overload* and *switch_off_overload*, with the former being similar to the *switch_on* task, but allowing the setting of the *power_on* predicate for an instrument even if the *power_avail* predicate is not true for the according satellite. We assume the *power_avail* predicate represents a restriction of maximum power for safety purposes that can be ignored. For this to work as intended, we additionally introduce the predicates *(overloaded ?arg1 - satellite)* and *(overloads ?arg0 - instrument ?arg1 - satellite)*, which denote if a satellite has currently a second instrument powered and what instrument is powered for a satellite. The task *switch_off_overload*, switches the overload using instrument and the normal power using instrument off. To give the planner the choice of using *overload*, we introduce two methods *method4_overload* and *method5_overload*, both resembling the similarly named methods *method4* and *method5*, with the difference of incorporating *overload*. With this, *method5_overload* represents an alternative to *method4*. Instead of turning an instrument off to power a different one, *method5_overload* allows the use of the *overloaded* predicate to achieve the same effect. However, this is the risk bearing choice. The task *switch_off_overload* turns power off for both instruments at the same cost as *switch_off* switching off one instrument, saving 15 seconds by skipping a plan step. On top of that, we introduce *superload* which is equivalent to *overload*, with all the same additional methods, tasks and predicates as *overload*, except it allows a third instrument to be activated after overload under the immense risk of inducing a prolonged power failure. With these changes, the planner can decide to cycle through instruments as originally intended, by switching power off and on every time, or skip *switch_off* tasks by using *overload* and *superload*, and eventually switching off power for all instruments at once.

5.2.1 Experiment

For the *Satellite-RA* domain, we present the influence of risk attitudes with the problem *3obs-1sat-3mod*[4]. This problem expects one satellite *satellite0* with three instruments *instrument01, instrument02* and *instrument03* to do one observation in each of the three directions *Phenomenon4, star0* and *Phenomenon6*. On top of that, each observation is to be made with a different mode of either *thermograph, x_ray* or *hd_video* and one instrument supports only one mode, making activation of all three instruments a necessity. All primitive tasks of the *Satellite-RA* domain have the *costdist* section attached, with a distribution of (1(15)), that guarantees a cost of 15 units. Exceptions are the *overload* and *superload* primitive tasks, which have distributions of (0.99(15) 0.01(100)) and (0.95(15)) (0.05(400)) respectively. With that, superload is riskier than overload, since the undesirable outcome costs more time and has a higher chance of occurrence.

The three primitive task sequences in Listing 5.4 represent the solution for each risk attitude. Comparison of these solutions yields a difference in the course of action between all risk attitudes. The planning with risk averse attitude results in a solution, without using *overload* or *superload*, while the risk neutral attitude results in the usage of only *overload* and the risk seeking attitude in usage of both. The amount of plan steps differ for each risk attitude as well, with 17, 16 and 15 for the risk averse, neutral and seeking attitudes, respectively. Exact differences can be observed in plan step five, where the risk neutral and seeking solutions embrace the risk *overload* brings, while the risk averse solution avoids it. Another difference is identifiable in plan step ten, where the risk neutral and risk averse solutions avoid *superload* while the risk seeking solution embraces it. With

[4]This problem instance can be viewed in the Appendix A.2

Listing 5.4 The primitive task sequences of three computed plans by RAPAND3 in the *Satellite-RA* domain. All plans are solutions for the problem instance *3obs-1sat-3mod*, and each solution is computed with a different risk attitude.

```
;; risk averse attitude, expected cost: 255; expected utility: -10^{9.78}
0 switch_on instrument03 satellite0
1 turn_to satellite0 GroundStation0 Phenomenon6
2 calibrate satellite0 instrument03 GroundStation0
3 turn_to satellite0 Phenomenon6 GroundStation0
4 take_image satellite0 Phenomenon6 instrument03 hd_video
5 switch_off instrument03 satellite0
6 switch_on instrument02 satellite0
7 turn_to satellite0 GroundStation0 Phenomenon6
8 calibrate satellite0 instrument02 GroundStation0
9 turn_to satellite0 Star5 GroundStation0
10 take_image satellite0 Star5 instrument02 x_ray
11 switch_off instrument02 satellite0
12 switch_on instrument01 satellite0
13 turn_to satellite0 GroundStation0 Star5
14 calibrate satellite0 instrument01 GroundStation0
15 turn_to satellite0 Phenomenon4 GroundStation0
16 take_image satellite0 Phenomenon4 instrument01 thermograph

;; risk neutral attitude, expected cost: 240.85, expected utility: -240.85
0 switch_on instrument03 satellite0
1 turn_to satellite0 GroundStation0 Phenomenon6
2 calibrate satellite0 instrument03 GroundStation0
3 turn_to satellite0 Phenomenon6 GroundStation0
4 take_image satellite0 Phenomenon6 instrument03 hd_video
5 overload instrument02 satellite0
6 turn_to satellite0 GroundStation0 Phenomenon6
7 calibrate satellite0 instrument02 GroundStation0
8 turn_to satellite0 Star5 GroundStation0
9 take_image satellite0 Star5 instrument02 x_ray
10 switch_off_overload instrument02 instrument02 satellite0
11 switch_on instrument01 satellite0
12 turn_to satellite0 GroundStation0 Star5
13 calibrate satellite0 instrument01 GroundStation0
14 turn_to satellite0 Phenomenon4 GroundStation0
15 take_image satellite0 Phenomenon4 instrument01 thermograph

;; risk seeking attitude, expected cost: 245.1, expected utility: 10^{-48.89}
0 switch_on instrument03 satellite0
1 turn_to satellite0 GroundStation0 Phenomenon6
2 calibrate satellite0 instrument03 GroundStation0
3 turn_to satellite0 Phenomenon6 GroundStation0
4 take_image satellite0 Phenomenon6 instrument03 hd_video
5 overload instrument02 satellite0
6 turn_to satellite0 GroundStation0 Phenomenon6
7 calibrate satellite0 instrument02 GroundStation0
8 turn_to satellite0 Star5 GroundStation0
9 take_image satellite0 Star5 instrument02 x_ray
10 superload instrument01 satellite0
11 turn_to satellite0 GroundStation0 Star5
12 calibrate satellite0 instrument01 GroundStation0
13 turn_to satellite0 Phenomenon4 GroundStation0
14 take_image satellite0 Phenomenon4 instrument01 thermograph
```

	Risk attitude	SL	Exp. cost	Exp. utility	Time
3obs-1sat-3mod	**Risk averse**	17	255.00	$-10^{9.77}$	3.76s
	Risk neutral	16	240.85	-150.85	3.94s
	Risk seeking	15	245.10	$10^{-48.88}$	3.26s
RA-4obs-1sat-3mod	**Risk averse**	23	345.00	$-10^{74.91}$	3.88s
	Risk neutral	18	270.85	-270.85	3.76s
	Risk seeking	17	275.10	$10^{-55.40}$	3.44s
RA-5obs-1sat-3mod	**Risk averse**	29	435.00	$-10^{94.46}$	6.89s
	Risk neutral	19	305.10	-305.10	5.82s
	Risk seeking	19	305.10	$10^{-61.91}$	5.55s
RA-6obs-1sat-3mod	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	-	–	–	-
RA-6obs-2sat-3mod	**Risk averse**	26	390.00	$-10^{84.69}$	2.91s
	Risk neutral	21	315.85	-315.85	3.01s
	Risk seeking	21	315.85	$10^{-68.41}$	3.00s
RA-4obs-1sat-4mod	**Risk averse**	23	345.00	$-10^{74.92}$	4.34s
	Risk neutral	21	316.70	-316.70	6.19s
	Risk seeking	21	316.70	$10^{-68.41}$	6.60s
RA-5obs-1sat-5mod	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	-	–	–	-
RA-5obs-2sat-5mod	**Risk averse**	28	420.00	$-10^{91.20}$	3.32s
	Risk neutral	26	391.70	-391.70	5.83s
	Risk seeking	25	395.95	$10^{-81.46}$	5.04s

Table 5.2: The results of running plan computation for the listed problem instances with RAPANDA3 in the *Satellite-RA* domain. A row refers to one problem instance and has three sub-rows associated, each referring to plan computation with the listed risk attitude. Solution length (SL) refers to the amount of plan steps the solution holds, with one plan step being a primitive task. Exp. cost displays the sum of individual expected costs of the plan steps. Exp. utility depicts the solution's expected utility, rounded to two decimal points, and the time column holds the planning time for the planner to compute the solution in seconds, rounded to two decimal points. Any computation that exceeded 15 minutes was aborted, and the cells are marked with -, for the problem instance exceeded our planning time limit.

this, the risk averse solution accepts a longer plan and a higher expected cost to minimize the risk it has to take. The risk seeking solution takes a chance at using *superload*, even though the expected cost advises against it.

We create additional problem instances, by increasing the scope and complexity of the showcased problem instance *3obs-1sat-3mod* in three different fashions. We increment the number of mandatory observations, the number of different modes available and the number of available satellites. The naming scheme of problem instances reveals the content of one instance, for example *3obs-1sat-3mod* reveals, that there are three observations needed, with one satellite and each observation is made with one of three different modes. Consequently, *5obs-2sat-4mod* means five observations are conducted with 2 satellites and at least two observations are made with the same mode. All satellites point initially in the same direction. What mode is necessary for what observation exactly can only be gleamed from looking at the problem instance file, which is too large to show here.

The results of our experiments are compiled into Table 5.2, where it appears as having six or more observations makes the plan computation overstep our time limit of 15 minutes. The same is true for having five or more different modes. Increasing the number of available satellites seems to relax the computational demand. The risk averse solution always possesses the most plan steps, due to how this domain is set up, with risky options being shortcuts. In contrast to the *Robot-RA* experiment, the risk neutral attitude's solution is not equal to the risk seeking solution for all problem instances, in terms of solution length and expected cost. That is a result of this domain providing effectively three choices when planning. Fully safe, not using *overload* or *superload*. Risky, using *overload*, but not *superload*. Extremely risky, using both *overload* and *superload*. With this, the risk neutral solution can take a middle ground and risk averse and risk seeking solutions may stray from this, depending on the intensity of the utility curve and the values of *costdist* sections. Consequently, the expected costs of risk seeking and risk averse solutions often differ from the optimal expected cost of the risk neutral attitude's solution. Interesting to examine are the problem instances in which the risk seeking solution agrees with the risk neutral one. It appears as that is the case, if the number of observations relative to the number of modes increases, which is likely due to the potential gains of *superload* becoming less relevant overall to the expected cost. However, *RA-4obs-1sat-4mod* presents an exception to our just mentioned statement. Most likely, a result of specific mathematical properties leading to almost no gain when choosing *superload*. Having four modes on four instruments means, four instruments need to be powered. In this case, choosing *superload* does not save a plan step as opposed to choosing *overload*. Using *overload* allows two instruments to be powered, and using *superload* allows three. Choosing either will lead to having to *switch_on* or *overload* another instrument, and the gain of using *switch_on* instead of *overload* is far outweighed by the loss of using *superload* instead of *overload*.

An additional interesting observation to make is that planning time does not differ much between risk attitudes, and a longer planning time does not seem to correlate with a longer solution. We believe that is the case since Satellite-RA problem instances set forth an initial task network consisting out of separately decomposable compound tasks, in contrast to the Robot-RA domain's problem instances.

5.3 Transport Domain

The *Transport* domain models a set of transporters taking various roads to pick up and unload packages at different locations.

Listing 5.5 The available lifted predicates in the *Transport-RA* domain.

```
(:predicates
  (road ?arg0 - location ?arg1 - location)
  (at ?arg0 - locatable ?arg1 - location)
  (in ?arg0 - package ?arg1 - vehicle)
  (capacity ?arg0 - vehicle ?arg1 - capacity_number)
  (capacity_predecessor ?arg0 - capacity_number ?arg1 - capacity_number)

  ;; risk-aware additions below
  (speedway ?arg0 - location ?arg1 - location)
)
```

The predicates in Listing 5.5 reveal the following types: *location*, *locatable*, *package*, *capacity_number*, *target*, *vehicle*, with *vehicle* and *package* being extensions of the *locatable* type. A pair of *location* can be connected via the *road* predicate, and every *locatable* can be associated with a *location*. Additionally, a *package* type is eligible to be in a vehicle through the predicate *in*. For our risk-aware addition, we introduce the *speedway* predicate, that connects two locations just like *road*. The *vehicle* type has a *capacity_number*, which can have multiple *capacity_predecessor*, representing available capacity in a *vehicle*. Every time a *vehicle* picks up a *package*, the preconditions check if there is a predecessor for the current *capacity_number*, allowing the pickup and setting the current capacity_number to the predecessor, if true. Unloading works the other way around.

The following paragraph is a description of the task hierarchy in the *Transport* domain, for which Figure 5.3 is an accompanying illustration. The domain possesses four compound tasks, *deliver*, *load*, *unload* and *get_to*. *Deliver* is the highest order task, representing one whole delivery. Problem instances usually have multiple *deliver* instances as initial task network. Consequently, *deliver* is decomposed into the rest of the abstract tasks in the sequence of *get_to*, *load*, *get_to*, *unload*. *Load* and *unload* have exactly one decomposition method each, resulting in the primitive tasks *pick_up* and *drop*, respectively, leaving no alternatives to a planner. In contrast, *get_to* has three alternatives. The first being *m_i_am_there_ordering_0*, which decomposes into the *noop* primitive task, an abbreviation for no operation and used if the planner decomposes into *get_to* if the vehicle is already at a desired *location*, for example if the truck is at the location of pickup but *deliver* has not been decomposed yet. The other two alternatives are *m_drive_to_ordering_0* and *m_drive_to_via_ordering_0*, with both decomposing into the same *drive* primitive task, although the latter additionally results in another *get_to* compound task, permitting recursion as a result. The *drive* task allows a vehicle to traverse one *road*, and with the recursive method, multiple roads to far away locations can be traversed.

To obtain *Transport-RA* from the *Transport* domain, we introduce two new methods, *m_drive_to_ordering_0_fast* and *m_drive_to_via_ordering_0_fast*, indicated through green borders in Figure 5.3. These new methods present a fast alternative to *m_drive_to_ordering_0* and *m_drive_to_via_ordering_0*, however, only locations connected via the *speedway* predicate allow for the fast alternative. In our problem instances, normal roads connect all locations in a corridor, while there is only one speedway connecting two locations. However, the speedway firstly leads to the speedway intersection, a special location that needs to be visited before the other speedway connected location can be reached. With this, the speedway constitutes a less direct way to desired locations, essentially giving a planner a choice of taking a risky, potentially faster but possibly

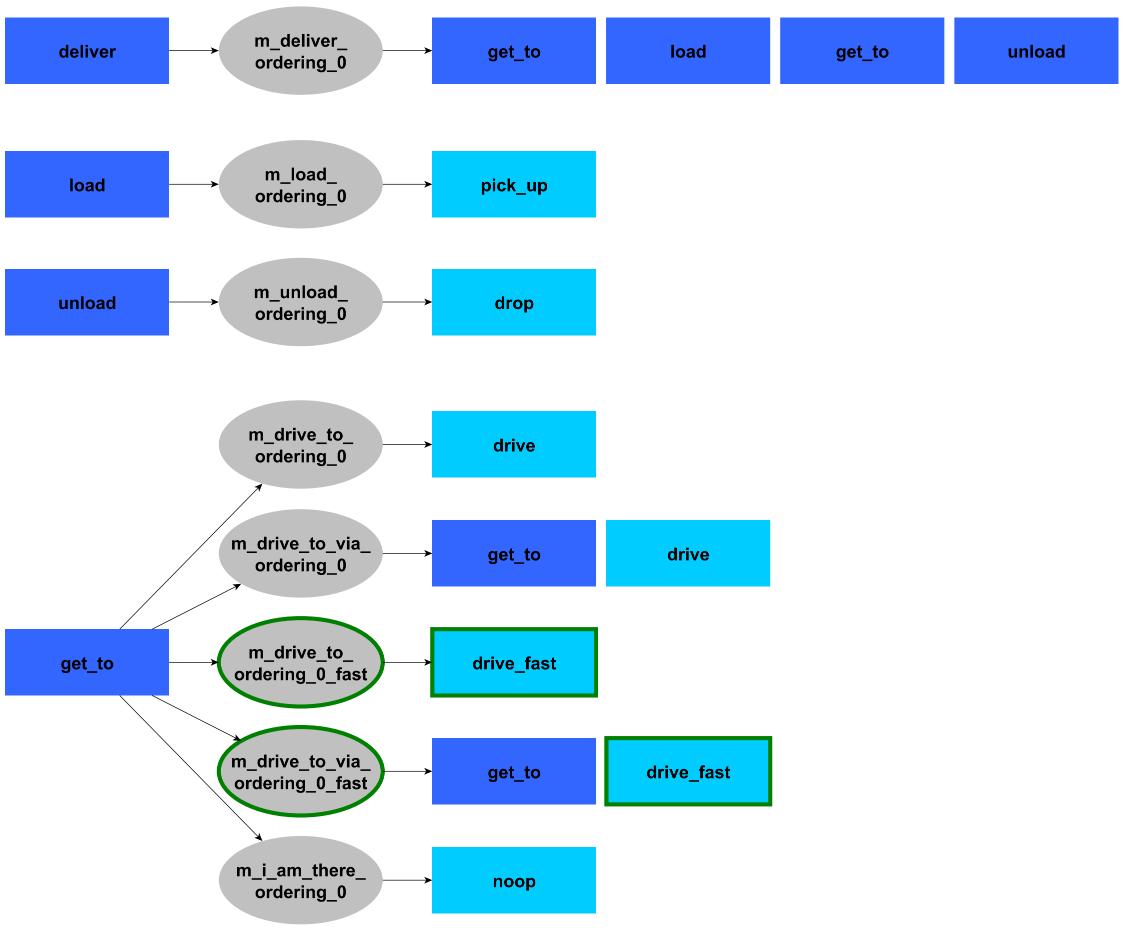


Figure 5.3: A diagram, illustrating the relations of methods and tasks in the *Transport-RA* domain. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks. Green borders indicate added content as part of the risk-aware adaption.

slower route. The new methods decompose into our new primitive task *drive_fast* and to reflect the risky but potentially faster route, this task receives the *costdist* section with a distribution of ((0.98(5)) (0.02(200))). All other primitive tasks likewise receive *costdist*, with a distribution of (1(15)), guaranteeing a 15 unit cost. The exception to this is the *noop* primitive task, which costs nothing, since it represents doing nothing.

5.3.1 Experiment

We show the influence of risk attitudes by planning for the problem instance *RA-3loc-2pack-1truck-speed01*[5] in the *Transport-RA* domain. The problem instance is a derivate of instance *pfile01*, a problem instance of the original domain. It is of similar nature, with the difference of including a

[5]The file can be viewed in the Appendix A.3

Listing 5.6 The primitive task sequences of three computed plans by RAPAND3 in the *Transport-RA* domain. All plans are solutions for the self created problem instance *RA-3loc-2pack-1truck-speed01* derived from an IPC 2020 provided problem instance, and each solution is computed with a different risk attitude.

```
;; risk averse attitude, expected cost: 120; expected utility: -10^{24.85}
0 noop truck_0 loc_2
1 drive truck_0 loc_2 loc_1
2 pick_up truck_0 loc_1 package_0 capacity_0 capacity_1
3 noop truck_0 loc_1
4 drive truck_0 loc_1 loc_0
5 drop truck_0 loc_0 package_0 capacity_0 capacity_1
6 noop truck_0 loc_0
7 drive truck_0 loc_0 loc_1
8 pick_up truck_0 loc_1 package_1 capacity_0 capacity_1
9 noop truck_0 loc_1
10 drive truck_0 loc_1 loc_2
11 drop truck_0 loc_2 package_1 capacity_0 capacity_1

;; risk neutral attitude, expected cost: 120; expected utility: -120
0 drive truck_0 loc_2 loc_1
1 pick_up truck_0 loc_1 package_0 capacity_0 capacity_1
2 drive truck_0 loc_1 loc_0
3 drop truck_0 loc_0 package_0 capacity_0 capacity_1
4 drive truck_0 loc_0 loc_1
5 pick_up truck_0 loc_1 package_1 capacity_0 capacity_1
6 drive truck_0 loc_1 loc_2
7 drop truck_0 loc_2 package_1 capacity_0 capacity_1

;; risk seeking attitude, expected cost: 125.6; expected utility: 10^{-23.92}
0 drive truck_0 loc_2 loc_1
1 pick_up truck_0 loc_1 package_0 capacity_0 capacity_1
2 drive_fast truck_0 loc_1 speedwayInters
3 drive_fast truck_0 speedwayInters loc_0
4 drop truck_0 loc_0 package_0 capacity_0 capacity_1
5 drive_fast truck_0 loc_0 speedwayInters
6 drive_fast truck_0 speedwayInters loc_1
7 pick_up truck_0 loc_1 package_1 capacity_0 capacity_1
8 drive truck_0 loc_1 loc_2
9 drop truck_0 loc_2 package_1 capacity_0 capacity_1
```

speedway between *loc_0* and *loc_1*. The problem presents one truck *truck_0*, three locations *loc_0, loc_1* and *loc_2*, two packages *package_0* and *package_1*, with *truck_0* having a capacity of one package. Both packages are situated at *loc_1* and *package_0* has to be delivered to *loc_0*, while *package_1* needs to end up in *loc_2*. Driving from *loc_0* to *loc_1*, presents the choice of taking the speedway, which results in a longer solution and holds high risk for potential high gains.

Shown in Listing 5.6 are three solutions computed for problem instance *RA-3loc-2pack-1truck-speed01*. The risk averse solution does not incorporate *drive_fast* at all. Additionally, it has the lowest expected cost, on par with the risk neutral solution. The risk seeking solution ends up having the highest expected cost, as it hopes to gain from driving fast on the speedway in plan steps two, three, five and six. This shows the planner embracing risk at higher expected cost, if informed by expected utility, derived from a risk seeking attitude's utility function. Interestingly, the risk averse solution is the same as the risk neutral one, except it incorporates the *noop* primitive task, which seems pointless, since *noop* costs nothing and does nothing. Consequently, this behaviour has no impact on risk consideration and *noop* tasks receive special treatment in following statistics by presenting the solution length with and without *noop* tasks.

For further experimentation, we designed additional problem instances by making various minor changes to *RA-3loc-2pack-1truck-speed01*. The naming scheme reveals the content of a problem instance. For example, *RA-3loc-2pack-1truck-speed01* expresses three locations, with two packages that need delivery, with one truck, making the deliveries. The first package always needs to be delivered to the last location, and all locations are connected via roads in a corridor. The string "speed" denotes the existence of a speedway between the location numbers, following the string. More detailed information about a problem instance, such as the starting location of trucks, is not included in this scheme.

The results of our experiments are compiled into Table 5.3. Looking at the table, it becomes clear that the showcased problem instance is already close to exceeding the complexity limit, induced by our planning time limit. Increasing the amount of locations by one, already makes risk seeking attitude's planning exceed our time limit. Relaxing the problem instance, by removing the second package, solves that. However, adding another location leads to the exceeding of our planning time limit by the risk seeking attitude's planning. Adding a conveniently placed truck to the *RA-4loc-2pack-1truck-speed12* problem instance, relaxes the problem enough for risk seeking attitude's computation to stay in time bounds, but adding another location results in the computation, exceeding our time limit again.

What can be discerned are the generally longer risk seeking solutions and the tendency of their computations to exceed our time limit of 15 minutes. This is not unexpected, since the domain is set up to allow for a choice between a longer and potentially less costing solution and a shorter, invariably costed solution. The risk seeking solution includes the speedway with the *drive_fast* primitive task, which requires an additional plan step compared to taking the road, prolonging computation as a result. In contrast to the *Robot-RA* domain, discussed in Section 5.1, this leads to the expected cost being in favour for the risk averse solution. Looking at solution length and expected cost, the risk neutral solution always seems to be the same as the risk averse one, if *noop* tasks are ignored. This is a result of the domain only offering two risk related choices. It becomes a question of how much expected cost is the risk seeking attitude willing to accept, for the chance of gains, from taking the longer speedway path. This depends on the specific values of the *costdist* section and the intensity of the utility function.

	Risk attitude	SL	Exp. cost	Exp. utility	Time
RA-3loc-2pack-1truck-speed01	**Risk averse**	8 (12)	120.00	$-10^{24.85}$	1.98s
	Risk neutral	8	120.00	-120.00	2.32s
	Risk seeking	10	125.60	$10^{-23.92}$	25.14s
RA-4loc-2pack-1truck-speed12	**Risk averse**	9 (13)	135.00	$-10^{28.11}$	2.61s
	Risk neutral	9	135.00	-135.00	2.32s
	Risk seeking	-	–	–	-
RA-4loc-1pack-1truck-speed12	**Risk averse**	5 (7)	75.00	$-10^{15.68}$	2.34s
	Risk neutral	5	75.00	-75.00	2.39s
	Risk seeking	7	80.60	$10^{-14.15}$	2.36s
RA-5loc-1pack-1truck-speed12	**Risk averse**	7 (9)	105.00	$-10^{22.20}$	2.60s
	Risk neutral	7	105.00	-105.00	14.87s
	Risk seeking	-	–	–	-
RA-4loc-2pack-2truck-speed12	**Risk averse**	8 (12)	120.00	$-10^{24.85}$	2.07s
	Risk neutral	8 (9)	120.00	-120.00	4.91s
	Risk seeking	11	140.60	$10^{-24.22}$	17.04s
RA-5loc-2pack-2truck-speed12	**Risk averse**	9 (13)	135.00	$-10^{28.11}$	2.75s
	Risk neutral	9 (10)	135.00	-135.00	63.77s
	Risk seeking	-	–	–	-

Table 5.3: The results of running plan computation for the listed problem instances with RAPANDA3 in the *Transport-RA* domain. A row refers to one problem instance and has three sub-rows associated, each referring to plan computation with the listed risk attitude. Solution length (SL) refers to the amount of plan steps the solution holds, with one plan step being a primitive task. Solutions in this domain might include *noop* tasks. We list Solution length including *noop* tasks in parentheses. Exp. cost displays the sum of individual expected costs of the plan steps. Exp. utility depicts the solution's expected utility, rounded to two decimal points, and the time column holds the planning time for the planner to compute the solution in seconds, rounded to two decimal points. Any computation that exceeded 15 minutes was aborted, and the cells are marked with -, for the problem instance exceeded our planning time limit.

5.4 Car-Fleet-RA Domain

Car-Fleet-RA is as cycle free, totally ordered domain and is fully self-designed for this work. It depicts an autonomous vehicle fleet, getting electric cars ready for their assignments, while dealing with limited space and equipment.

Looking at the predicates in Listing 5.7, four types are discernible, *car*, *depot*, *equipment* and *assignment*. Additional types are: *cleaning-equip*, *maintaining-equip* and *reloading-equip*, with all being extensions of *equipment*. The predicate *awaits-collection* is one of many predicates that

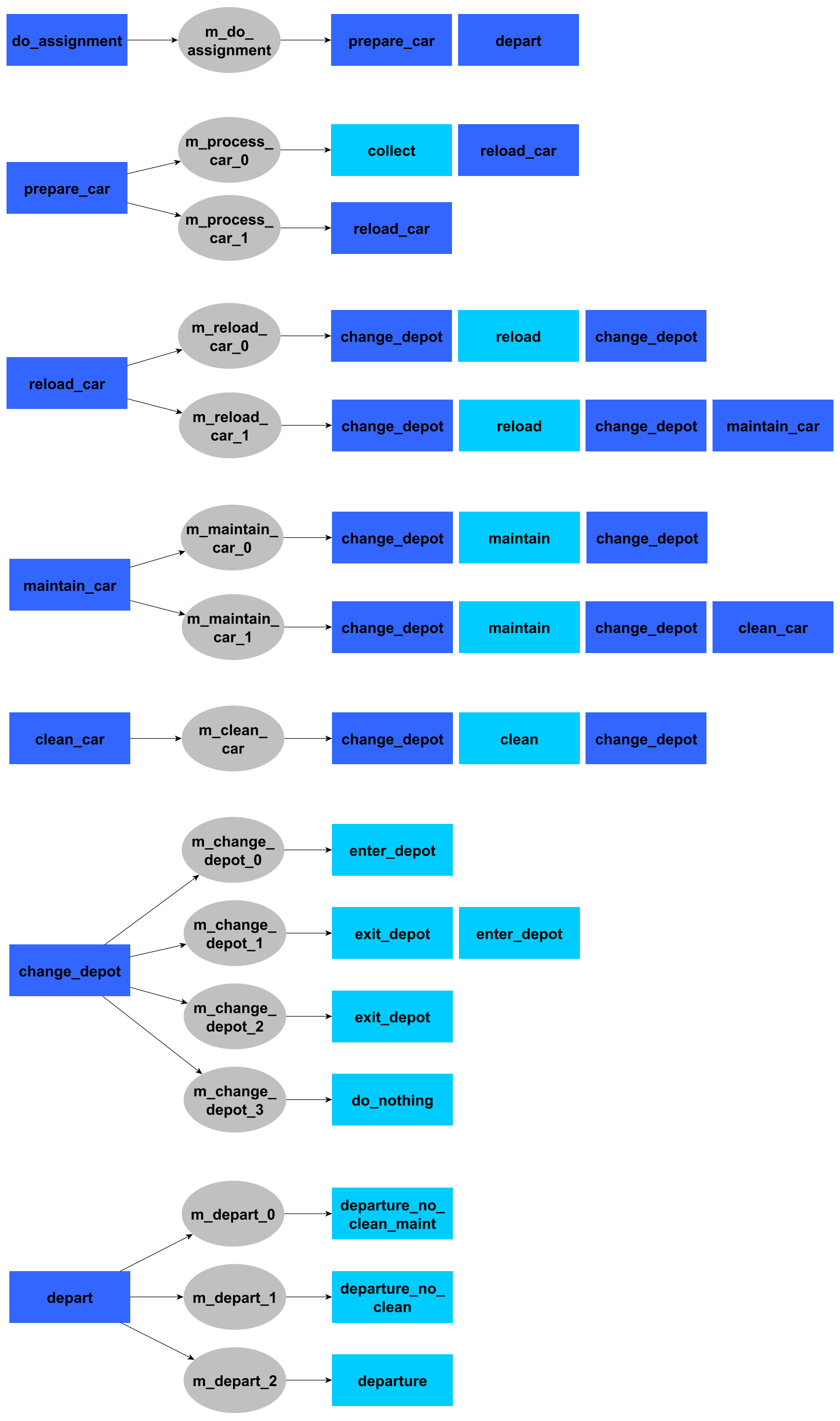


Figure 5.4: A diagram, illustrating the relations of methods and tasks in the *Car-Fleet-RA* domain. Grey ovals represent methods, light blue rectangles primitive tasks and dark blue rectangles compound tasks.

Listing 5.7 The available lifted predicates in the *Car-Fleet-RA* domain.

```
(:predicates
    (awaits-collection ?c - car)
    (on-staging-area ?c - car)
    (in-depot ?c - car)
    (departed ?c - car)
    (depot-free ?d - depot)
    (staging-area-free)
    (staging-area-content ?c - car)
    (car-in ?d - depot ?c - car)
    (has-equipment ?d - depot ?g - equipment)
    (assignment-done ?a - assignment)
    (car-clean ?c - car)
    (car-maintained ?c - car)
    (car-loaded ?c - car)
)
```

allow a *car* to be in various conditions, for example it can be waiting to be collected by the car fleet AI agent, or be clean or loaded, represented by *car-clean* and *car-loaded*, respectively. On top of that, with the predicates *depot-free* and *has-equipment*, a *depot* can hold exactly one or no *car* and posses zero or multiple *equipment*, needed for cleaning, maintaining or loading a *car*. Cars can only be maintained, cleaned or loaded in depots, possessing the matching equipment. All cars departing for assignments or changing depots have to go through the staging area. There can be only one *car* at once on the staging area. The predicate *staging-area-free* models this restriction. The ultimate goal of this domain is to compute a plan, with multiple *do_assignment* compound tasks in the initial task network. *Do_assignment* starts the process of preparing a *car* and making it depart for assignment. Preparing a *car* includes loading, maintaining and cleaning it. Any *car* can be sent on any assignment, as long as it is at least loaded. Maintaining and cleaning is optional for risk demonstration purposes, enabling a planner to decide to forgo preparation steps under risk of high cost. Higher cost outcomes represent the chance of dissatisfaction on the customer side due to the lack of cleanliness, or the chance of a failure of operation due to skipping maintenance.

Figure 5.4 shows the task hierarchy in the Car-Fleet domain and is reference for the explanation in the following paragraph. The initial task network consists of one or multiple *do_assignment* compound tasks, which are decomposed to *prepare_car* and *depart*. The task *prepare_car* can either be decomposed to *collect* and *reload_car* or only *reload_car*, with the latter option reflecting the car being on the premise already and not in need of collection, therefore there is no alternative present, the method selection solely depends on the predicate *awaits-collection*. The compound tasks *reload_car*, *maintain_car* and *clean_car* share several similarities and represent the preparation steps of a car. They all decompose into the corresponding primitive task, such as *reload*, flanked by *change_Depot*. The tasks *reload_car* and *maintain_car* additionally induce a choice between two alternatives. Decomposition of both can either include or omit the next preparation compound task, giving the planner the choice to either commit to one more preparation step or to stop preparation now. This choice is where risk is portrayed in this domain. Fully preparing is more expensive, but leads to less risk later in the car's departure. It should be noted, that a car can not be unmaintained but clean, due to how the chain of preparation compound tasks is set up. Flanking the preparation primitive tasks with *change_depot* compound tasks leads to a high number of said compound tasks.

Having many *change_Depot* tasks in partial plans gives a planner the necessary opportunity to rearrange the location of the car being prepared, with an appropriate binding of the parameter variables, since not all depots are free or equipped adequately. This is achieved by decomposing the *change_depot* into four optional task sequences, with some being alternatives to each other. The options include the primitive tasks *enter_depot*, *exit_depot* and *do_nothing* in several forms. Despite that, these do not relate to risk and only serve to get the cars to an appropriate depot. Finally, the *depart* compound task is decomposed into a primitive task representing the state of the car depending on the preparations made. If the predicates indicate, that for a given car cleaning and maintaining was skipped, then the planner has to decompose into *departure_no_clean_maint*, which has the highest risk associated but fulfils the assignment predicate nevertheless. This works analogously for the two other options, *departure_no_clean* and *departure*.

5.4.1 Experiment

Computing solutions for problem instance *RA-a1-car2-dep3*[6], illustrates how risk attitudes affect planning, as can be seen in Listing 5.8. The problem defines one assignment *a0*, two cars *car0* *car1* and three depots *depot0, depot1* and *depot2*, with *depot0* containing reloading equipment, *depot1* cleaning equipment and *depot2* maintaining equipment. Additionally, *car0* is stationed in *depot0* and *car1* is not on the premise and awaits collection. All primitive tasks are guaranteed to cost 15 units, possessing the *costdist* section, with a distribution of (1(15)). There are three exceptions, however. The task *do_nothing* has no cost, while the task *departure_no_clean* and *departure_no_clean_maint* have risk associated, with distributions of ((0.5(15)) (0.5(100))) and (0.45(15)) (0.5(100)) (0.05(1000))) respectively. A solution for each risk attitude is shown in Listing 5.8, where striking differences are noticeable. The risk averse planning produced a solution, that makes the car depart with cleanliness and maintenance even though, as shown by the risk neutral solution, this is not the expected cost optimal plan. While risk neutrality makes a car depart without cleaning, risk aversion avoids this risk as expected. Detailed comparison between the risk averse and neutral solutions show differences in length and the neutral solution not using the *clean* task. The risk neutral solution also differs from the risk seeking one, where the risk seeking solution lets a car depart with no maintenance and cleaning. Since this leads to a higher expected cost compared to risk neutrality, the risk seeking attitude takes a risk, even if one could argue for its inadvisability.

We incrementally made various minor changes to *RA-a1-car2-dep3*, resulting in additional problem instances for us to run on RAPANDA3. The naming scheme reveals the content of the problem instance. For example, *RA-a1-car2-dep3* denotes that there is one assignment to fulfil, with two cars and three depots available. The equipment of the depots is not shown by the naming scheme, but it can be assumed that each of the three equipment types is present at least once. Unless stated otherwise, the initial position of cars is always not on site awaiting to be collected except for *car0* which is stationed in *depot0*.

The results of our experiments are compiled into Table 5.4. There are not many entries, since the domain is rather intricate, and expanding the original problem instance quickly leads to overstepping our planning time limit. From what we have, one can observe increasing the number of assignments

[6]The problem instance can be viewed in the Appendix A.4

Listing 5.8 The primitive task sequences of three computed plans by RAPAND3 in our self designed *Car-Fleet-RA* domain. All plans are solutions for the problem *RA-a1-car2-dep3*, and each solution is computed with a different risk attitude.

```
;; risk averse attitude, expected cost: 135; expected utility: -10^{28.41}
0 do_nothing car0
1 reload car0 depot0 reloading0
2 __method_precondition_m_change_depot_1_9_precondition car0 depot2
3 exit_depot depot0 car0
4 enter_depot depot2 car0
5 do_nothing car0
6 maintain car0 depot2 maintaining0
7 do_nothing car0
8 __method_precondition_m_change_depot_1_9_precondition car0 depot1
9 exit_depot depot2 car0
10 enter_depot depot1 car0
11 clean car0 depot1 cleaning0
12 __method_precondition_m_change_depot_2_10_precondition car0
13 exit_depot depot1 car0
14 departure car0 a0

;; risk neutral attitude, expected cost: 132.5; expected utility: -132.5
0 do_nothing car0
1 reload car0 depot0 reloading0
2 __method_precondition_m_change_depot_2_10_precondition car0
3 exit_depot depot0 car0
4 __method_precondition_m_change_depot_0_8_precondition car0
5 enter_depot depot2 car0
6 maintain car0 depot2 maintaining0
7 __method_precondition_m_change_depot_2_10_precondition car0
8 exit_depot depot2 car0
9 departure_no_clean car0 a0

;; risk seeking attitude, expected cost: 136.75; expected utility: 10^{-10.42}
0 do_nothing car0
1 reload car0 depot0 reloading0
2 __method_precondition_m_change_depot_2_10_precondition car0
3 exit_depot depot0 car0
4 departure_no_clean_maint car0 a0
```

to three or higher is making risk averse and risk neutral computation overstep our time limit. Even after simplifying the problem, with *RA-a3-car3-dep3-moreEquip* giving more depots a greater variety of equipment, no solution under 15 minutes is found. Risk averse and neutral computation being the once that fail is likely a result of generally longer solutions for these attitudes, as taking risk in this domain is about generating a shorter solution by skipping either maintenance or maintenance and cleaning. In this domain, risk neutrality can take a middle ground, computing to a solution which is a compromise between risk averse and seeking. This can be observed in Listing 5.8 and Table 5.4, by considering the differences in solution lengths.

	Risk attitude	SL	Exp. cost	Exp. utility	time
RA-a1-car2-dep3	**Risk averse**	9 (12)	135.00	$-10^{28.41}$	2.34s
	Risk neutral	6 (7)	132.50	-132.50	2.08s
	Risk seeking	3 (4)	136.75	$10^{-10.42}$	1.96s
RA-a2-car2-dep3	**Risk averse**	20 (25)	300.00	$-10^{63.64}$	46.6s
	Risk neutral	14 (16)	295.00	-295.00	34.47s
	Risk seeking	8 (9)	303.50	$10^{-27.05}$	2.24s
RA-a3-car3-dep3	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	13 (14)	470.25	$10^{-43.69}$	2.24s
RA-a3-car3-dep3-moreEquip	**Risk averse**	-	–	–	-
	Risk neutral	-	–	–	-
	Risk seeking	13 (14)	470.25	$10^{-43.69}$	17.98s

Table 5.4: The results of running plan computation for the listed problem instances with RAPANDA3 in the *Car-Fleet-RA* domain. A row refers to one problem instance and has three sub-rows associated, each referring to plan computation with the listed risk attitude. Solution length refers to the amount of plan steps the solution holds, with one plan step being a primitive task. Solutions in this domain might include *do_nothing* tasks. We list solution length (SL) including *do_nothing* tasks in parentheses. Exp. cost displays the sum of individual expected costs of the plan steps. Exp. utility depicts the solution's expected utility, rounded to two decimal points, and the time column holds the planning time for RAPANDA3 to compute the solution in seconds, rounded to two decimal points. Any computation that exceeded 15 minutes was aborted, and the cells are marked with -, for the problem instance exceeded our planning time limit.

6 Related Work

This work draws inspiration from [AGA22b], where risk-aware plan-based HTN planning has been proposed. Similar to the approach examined in our work, the authors assume multiple possible costs for a primitive task. Furthermore, they employ utility functions based on risk attitudes to heuristically guide planning, although the functions differ somewhat to the functions we use. Another difference is that we provide an implementation and experimental evaluation of the theoretical approach. Such an experimental evaluation is conducted in [Smi23] as well, providing an outlook on the effects of Risk-aware HTN planning for a risk aware domain. Similar to our work, utilities representing risk attitudes are calculated to be used as heuristic for HTN planning. An implementation of risk-aware HTN planning is provided and examined with an HTN domain. However, the authors propose a state-based risk-aware HTN planning approach, while we describe a plan-based risk-aware HTN planning approach.

In Bercher, Behnke, Höller et al. [BBHB17], the authors introduce two heuristics for HTN planning, namely $\mathrm{TDG_c}$ and TDG_m, both relying on the TDG induced by HTN domains. While the admissible heuristic $\mathrm{TDG_c}$ guides the planning computation by minimizing over summed up task costs, $\mathrm{TDG_m}$ aggregates the number of decompositions and causal link insertions needed and minimizes over these. $\mathrm{TDG_c}$ is similar to the risk-aware plan-based HTN approach we implement, since it associates tasks with costs and exploits the TDG to compute cost estimates of compound tasks and decomposition methods. Even so, their work only considers single task costs, which makes the proposed heuristics only applicable to deterministic domains. In contrast, risk-aware plan-based HNT planning assigns variable task costs, to model non-deterministic domains. Providing guidance in non-deterministic domains is possible in [TMSP11] through usage of Markov decision process (MDP). The Authors propose a method for converting HTNs to Earley graphs, which then can be evaluated, similarly to MDP, to obtain the optimal plan by calculating maximum expected utility, with utility defined as the maximum expected reward from MDP. This approach annotates methods with probabilities, while our approach undertakes this for primitive tasks. On top of that, our work implements a heuristic in a proof of concept, with which the heuristic is tested.

We provide various probabilistic domains, which model different outcomes of primitive tasks by means of a probability distribution over possible costs. Multiple other works concern themselves with similar probabilistic domains. The work in [HKM09] deals with the problem of inaccurate probabilities for outcomes, by proposing a learning approach to the creation of HTN domains. A learning system is provided, which creates domain knowledge for HTN domains with multiple outcomes. While parts of our work deal with transforming deterministic domains into non-deterministic ones, our added domain knowledge is handwritten and not learned by means of an algorithm. Furthermore, the authors of Hogg, Kuter, and Munoz-Avila [HKM09] do not implement their own heuristic and changes to an existing planner, rather they experiment with an already existing one.

In [LCKY09], techniques of plan recognition are applied to learn and model user preferences on plans, where probabilities are associated with decomposition methods. Similarly, our work concerns itself with user preferences, however these are expressed through utility functions. On top of that, we associate probabilities with primitive tasks instead of methods.

The authors of Mugan and Kuipers [MK11] present an approach to allow agents to solve problems by observing their real life environment. This utilizes dynamic Bayesian networks (DNBs) to capture possible contingencies of real life variables as actions, which results in learning a hierarchy of actions, which is comparable to HTNs used in our work, if one considers actions as tasks. In their approach, actions posses a reliability, denoting the chance of successful execution, thus depicting probability in a similar vain to our work. Yet, we do not consider success or failure as possible outcomes, rather we associate varying costs to primitive tasks.

The work of Morisset and Ghallab [MG08] defines HTNs as skills for a robot to overcome high level problems. Primitive tasks in this case are the most basic motor functions of the robot. Each problem has multiple possible skills for solving. The choice on which skill to use is made by taking environmental variables into account as state space in an MDP, while the skills constitute the action space. This induces a stochastic mapping from environmental variables to skills, effectively associating probabilities to HTNs, reflecting their suitability for the current environment. HTNs are chosen by the MDP optimization objective, seeking to maximize some cumulative function of rewards. Since HTNs are usually comprised of one high level compound task, the approach of Morisset and Ghallab [MG08] associates probabilities to these tasks similar to our work, except we associate with primitive tasks. Furthermore, we employ maximization of an expected function value as well, albeit utility instead of reward.

As we enrich domains with variable action costs and choice to examine RAPANDA3, we bring them closer to authentically representing non-deterministic real-world domains. With the same goal in mind, the authors of Alnazer, Georgievski, and Aiello [AGA22a] examine HTN domains and problems used in IPC 2020, in terms of realism by use of multiple criteria referred to as *realistic domain aspects*. One aspect being the number and types of method choices presented in a domain. The authors modify two ICP 2020 domains, bringing them closer to represent realistic domain aspects. Analogous to their work, modifications to ICP 2020 domains are designed in our work, with the different intention of using them as benchmark, but a similar result.

7 Conclusion and Outlook

We showed an approach to risk-aware plan-based HTN planning, which employs utility functions to heuristically guide plan-based HTN planning towards solutions, that adhere to a specified risk attitude. In the form of RAPANDA3, we implemented a proof of concept to this approach. We showed in an experimental setting, that RAPANDA3 indeed chooses a different solution for different risk attitudes, with the solution, adhering to a specified risk attitude. Additionally, we discussed how specific domain properties impact the risk attitudes' solutions. However, we discovered that planning with intricate problem instances on RAPANDA3 is resource demanding. So much so, that we could not get results with many IPC 2020 benchmark problem instances. After consulting with developers of PANDA3, Gregor Behnke and Pascal Bercher, we believe the reasons for these performance issues being the following. PANDA3 itself is resource intensive, and plan-based HTN planning in general is resource intensive. We believe PANDA3 is an issue, as the original developers moved on from PANDA3 two years ago and so PANDA3 has not received implementations of advancements in pruning techniques that speed up planning. We believe plan-based HTN planning to be an issue, since in IPC 2023, PYHIPOP, the only competing plan-based HTN planner, performed considerably worse than all other participants [IPCb; LA21]. Without performance improvements, risk-aware plan-based HTN planning is not suitable for industry. To move away from plan-based HTN planning could improve performance, however, this would require a completely different approach than risk-aware plan-based HTN planning. When moving away from plan-based HTN planning, we think it advisable to implement risk-aware planning on a newer planner, such as participants of IPC 2023.

The risk-aware plan-based HTN planning approach relies on a probability distribution over costs for different outcomes of primitive tasks. We assume that different outcomes always have the same effect, however it is plausible to have different effects for different outcomes as well. In future work, this could be addressed by finding a way to have the state change, depending on outcome.

Besides implementing RAPANDA3, we proposed the costdist section for the HDDL language definition, which allows representing a probability distribution over costs for a primitive task. The necessary grammar and parsing, for the costdist section to be recognized correctly, is implemented in RAPANDA3. The costdist section only models cost and probability for lifted primitive tasks. A way to enhance the costdist section, is to make it applicable to instances of tasks as well. Currently, regardless of the constants bound to the parameter variables of a task, a task always has the same probability distribution of costs. Imagine a domain with a road type and multiple road constants. Having the ability to attach costdist to a road constant, which in turn imposes this probability distribution on task instances, having said road bound to one of its parameter variables, allows for a convenient way to define roads of varying risks, without having to define multiple primitive tasks.

Lastly, we introduced multiple new HTN domains and problem instances, in the HDDL language format. Three of these domains are based on already existing ones and one is fully original. They all share the attribute of being risk-aware, meaning they use the costdist section to have a probability distribution over costs for primitive tasks. On top of that, these domains are designed to create freedom of choice for planners, so that risk is able of being taken in the first place. A reasonable avenue for future work, concerning risk-aware plan-based HTN planning, is the creation of additional risk-aware domains and corresponding problem instances.

A Appendix

A.1 Robot-RA

```
(define
 (problem pfile_RA01)

 (:domain robot)

(:objects o1 o2 - PACKAGE c r1 r2 r3 - ROOM d01 d12 d23 - ROOMDOOR)
 (:htn
  :ordered-tasks (and
    (task0 (achieve-goals))
  )
 )
(:init
(rloc c)
(armempty)
(door c r1 d01)
(door r1 r2 d12)
(door r2 r3 d23)
(door r1 c d01)
(door r2 r1 d12)
(door r3 r2 d23)
(closed d23)
(in o1 c)
(in o2 r2)
(goal_in o1 r3) (goal_in o2 r1))

 (:goal (and
  (in o1 r3)
  (in o2 r1)
    ))
)
```

Listing A.1: The HDDL file of the problem instance pfile_RA01

A.2 Satellite

```
(define
  (problem p3obs_1sat_3mod)
  (:domain  satellite2)
  (:objects
    GroundStation0 - calib_direction
    Phenomenon7 - image_direction
    Star5 - image_direction
    Phenomenon4 - image_direction
    Phenomenon8 - image_direction
    Phenomenon6 - image_direction
    instrument01 - instrument
    instrument02 - instrument
    instrument03 - instrument
    thermograph - mode
    x_ray - mode
    hd_video - mode
    satellite0 - satellite
  )
  (:htn
    :parameters ()
    :subtasks (and
     (task0 (do_observation Phenomenon4 thermograph))
     (task1 (do_observation Star5 x_ray))
     (task2 (do_observation Phenomenon6 hd_video))
    )
    :ordering (and
      (task1 < task0)
      (task2 < task1)
    )
  )
  (:init
    (on_board instrument01 satellite0)
    (supports instrument01 thermograph)
    (calibration_target instrument01 GroundStation0)
    (on_board instrument02 satellite0)
    (supports instrument02 x_ray)
    (calibration_target instrument02 GroundStation0)
    (on_board instrument03 satellite0)
    (supports instrument03 hd_video)
    (calibration_target instrument03 GroundStation0)
    (power_avail satellite0)
    (pointing satellite0 Phenomenon6)
  )
)
```

Listing A.2: The HDDL file of the problem instance 3obs-1sat-3mod

A.3 Transport-RA

```
(define
  (problem RA-3loc-2pack-1truck-speed01)
  (:domain  Transport-RA)
  (:objects
    package_0 - package
    package_1 - package
    capacity_0 - capacity_number
    capacity_1 - capacity_number
    loc_0 - location
    loc_1 - location
    loc_2 - location
    speedwayInters - location
    truck_0 - vehicle
  )
  (:htn
    :parameters ()
    :subtasks (and
     (task0 (deliver package_0 loc_0))
     (task1 (deliver package_1 loc_2))
    )
    :ordering (and
      (< task0 task1)
    )
  )
  (:init
    (capacity_predecessor capacity_0 capacity_1)
    (road loc_0 loc_1)
    (road loc_1 loc_0)
    (road loc_1 loc_2)
    (road loc_2 loc_1)
    (speedway loc_0 speedwayInters)
    (speedway speedwayInters loc_0)
    (speedway loc_1 speedwayInters)
    (speedway speedwayInters loc_1)
    (at package_0 loc_1)
    (at package_1 loc_1)
    (at truck_0 loc_2)
    (capacity truck_0 capacity_1)
  )
)
```

Listing A.3: The HDDL file of the problem instance RA-3loc-2pack-1truck-speed01

A.4 Car-Fleet-RA

```
(define
  (problem RA-a1-car2-dep3)
  (:domain Car-Fleet)

;---------------- Facts -----------------------
  (:objects
    a0 - assignment
    car0 - car
    car1 - car
    depot0 - depot
    depot1 - depot
    depot2 - depot
    reloading0 - reloading-equip
    cleaning0 - cleaning-equip
    maintaining0 - maintaining-equip
  )

;--------------- Initial State -----------------
  (:htn
    :parameters ()
    :ordered-subtasks (and
      (task1  (do_assignment a0))
    )
  )
  (:init
    (in-depot car0)
    (awaits-collection car1)
    (staging-area-free)
    (car-in depot0 car0)
    (depot-free depot1)
    (depot-free depot2)
    (has-equipment depot0 reloading0)
    (has-equipment depot1 cleaning0)
    (has-equipment depot2 maintaining0)
  )

  (:goal (and
    (assignment-done a0)
    )
  )
)
```

Listing A.4: The HDDL file of the problem instance RA-a1-car2-dep3

Bibliography

[AGA22a] E. Alnazer, I. Georgievski, M. Aiello. “On bringing HTN domains closer to reality-the case of satellite and rover domains”. In: *International Conference on Automated Planning Systems (ICAPS) Workshop on Scheduling and Planning Applications (SPARK)*. 2022 (cit. on p. 76).

[AGA22b] E. Alnazer, I. Georgievski, M. Aiello. *Risk Awareness in HTN Planning*. 2022. arXiv: 2204.10669 [cs.AI] (cit. on pp. 18, 33, 75).

[BAH19] P. Bercher, R. Alford, D. Höller. “A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations.” In: *IJCAI*. 2019, pp. 6267–6275 (cit. on p. 21).

[BBHB17] P. Bercher, G. Behnke, D. Höller, S. Biundo. “An Admissible HTN Planning Heuristic.” In: *IJCAI*. 2017, pp. 480–488 (cit. on pp. 18, 25, 30, 57, 75).

[BHBB19] G. Behnke, D. Höller, P. Bercher, S. Biundo. “More Succinct Grounding of HTN Planning Problems–Preliminary Results”. In: (2019) (cit. on p. 45).

[DBIG14] F. Dvorak, A. Bit-Monnot, F. Ingrand, M. Ghallab. “A flexible ANML actor and planner in robotics”. In: *Planning and Robotics (PlanRob) Workshop (ICAPS)*. 2014 (cit. on p. 18).

[EBSB12] M. Elkawkagy, P. Bercher, B. Schattenberg, S. Biundo. “Improving hierarchical planning performance by the use of landmarks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 26. 1. 2012, pp. 1763–1769 (cit. on p. 25).

[FN93] R. E. Fikes, N. J. Nilsson. “STRIPS, a retrospective”. In: *Artificial intelligence* 59.1-2 (1993), pp. 227–232 (cit. on p. 21).

[GA15] I. Georgievski, M. Aiello. “HTN planning: Overview, comparison, and beyond”. In: *Artificial Intelligence* 222 (2015), pp. 124–156 (cit. on pp. 21, 25, 27).

[GNN+17] I. Georgievski, T. A. Nguyen, F. Nizamic, B. Setz, A. Lazovik, M. Aiello. “Planning meets activity recognition: Service coordination for intelligent buildings”. In: *Pervasive and Mobile Computing* 38 (2017), pp. 110–139 (cit. on p. 18).

[GTFM13] A. González-Ferrer, A. Ten Teije, J. Fdez-Olivares, K. Milian. “Automated generation of patient-tailored electronic care pathways by translating computer-interpretable guidelines into hierarchical task networks”. In: *Artificial intelligence in medicine* 57.2 (2013), pp. 91–109 (cit. on p. 18).

[HBB+19] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, R. Alford. “HDDL–A Language to Describe Hierarchical Planning Problems”. In: *arXiv preprint arXiv:1911.05499* (2019) (cit. on p. 22).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA framework for hierarchical planning”. In: *KI-Künstliche Intelligenz* (2021), pp. 1–6 (cit. on pp. 18, 41).

[HKM09] C. Hogg, U. Kuter, H. Munoz-Avila. “Learning Hierarchical Task Networks for Nondeterministic Planning Domains.” In: *IJCAI*. 2009, pp. 1708–1714 (cit. on p. 75).

[IPCa] *International Planning Competition (IPC) on HTN Planning*. URL: https://ipc2020.hierarchical-task.net/ (cit. on pp. 51, 57).

[IPCb] *Results of the International Planning Competition (IPC) on HTN Planning in 2020*. URL: https://ipc2020.hierarchical-task.net/results/results (cit. on p. 77).

[KBK+07] J.-P. Kelly, A. Botea, S. Koenig, et al. “Planning with hierarchical task networks in video games”. In: *Proceedings of the ICAPS-07 Workshop on Planning in Games*. Citeseer. 2007 (cit. on p. 18).

[KBK08] J.-P. Kelly, A. Botea, S. Koenig. “Offline planning with hierarchical task networks in video games”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 4. 1. 2008, pp. 60–65 (cit. on p. 18).

[LA21] C. Lesire, A. Albore. “PYHIPOP-Hierarchical Partial-Order Planner”. In: *Workshop on the International Planning Competition*. 2021 (cit. on p. 77).

[LCKY09] N. Li, W. Cushing, S. Kambhampati, S. Yoon. “Learning user plan preferences obfuscated by feasibility constraints”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 19. 2009, pp. 370–373 (cit. on p. 76).

[MAMO18] J. Munoz-Morera, F. Alarcon, I. Maza, A. Ollero. “Combining a hierarchical task network planner with a constraint satisfaction solver for assembly operations involving routing problems in a multi-robot context”. In: *International Journal of Advanced Robotic Systems* 15.3 (2018), p. 1729881418782088 (cit. on p. 18).

[MG08] B. Morisset, M. Ghallab. “Learning how to combine sensory-motor functions into a robust behavior”. In: *Artificial intelligence* 172.4-5 (2008), pp. 392–412 (cit. on p. 76).

[MJC14] A. Menif, É. Jacopin, T. Cazenave. “SHPE: HTN Planning for Video Games”. In: *CGW@ECAI*. 2014. URL: https://api.semanticscholar.org/CorpusID:8224598 (cit. on p. 18).

[MK11] J. Mugan, B. Kuipers. “Autonomous learning of high-level states and actions in continuous environments”. In: *IEEE Transactions on Autonomous Mental Development* 4.1 (2011), pp. 70–86 (cit. on p. 76).

[NAI+03] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, F. Yaman. “SHOP2: An HTN planning system”. In: *Journal of artificial intelligence research* 20 (2003), pp. 379–404 (cit. on p. 18).

[NS61] A. Newell, H. A. Simon. “GPS, a program that simulates human thought”. In: (1961) (cit. on p. 21).

[PANDA23] *PANDA Description*. 2023. URL: https://panda-planner-dev.github.io/ (cit. on p. 41).

[SKP+14] R. Stern, S. Kiesel, R. Puzis, A. Felner, W. Ruml. “Max is more than min: Solving maximization problems with heuristic search”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 5. 1. 2014, pp. 148–156 (cit. on p. 37).

[Smi23] R. Smith. “Risk awareness in poker planning agents”. B.S. thesis. 2023 (cit. on p. 75).

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau. “HTN planning for web service composition using SHOP2”. In: *Journal of Web Semantics* 1.4 (2004), pp. 377–396 (cit. on p. 18).

[TMSP11] Y. Tang, F. Meneguzzi, K. Sycara, S. Parsons. “Probabilistic hierarchical planning over MDPs”. In: *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*. 2011, pp. 1143–1144 (cit. on p. 75).

[War76] D. H. Warren. “Generating conditional plans and programs”. In: *Proceedings of the 2nd Summer Conference on Artificial Intelligence and Simulation of Behaviour*. 1976, pp. 344–354 (cit. on p. 21).

[WK15] D. A. Wood, R. Khosravanian. “Exponential utility functions aid upstream decision making”. In: *Journal of Natural Gas Science and Engineering* 27 (2015), pp. 1482–1494 (cit. on p. 34).

[ZZY+14] W. Zeng, H. Zhou, M. You, et al. “Risk-sensitive multiagent decision-theoretic planning based on MDP and one-switch utility functions”. In: *Mathematical Problems in Engineering* 2014 (2014) (cit. on p. 18).

All links were last followed on October 26, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Fellbach, 26.10.2023, [illegible]

place, date, signature