

*MIDO : Mathématique et Informatique de la Décision et des organisations.*

**Maitres de stage :**

*Charles Rich*

*Nicolas Sabouret*

**Plan repair in reactive HTNs with incomplete model**

Table des matières :

[I. Introduction 2](#_Toc391371853)

[II. Background 3](#_Toc391371854)

[II.1. Classical and linear planning 3](#_Toc391371855)

[II.2. HTN planning 3](#_Toc391371856)

[Overview of HTN: 4](#_Toc391371857)

[II.3. Reactive HTNs: 6](#_Toc391371858)

[II.4. Plan execution, breakdown and recovery: 6](#_Toc391371859)

[II.5. the problem 7](#_Toc391371860)

[III. Related work 7](#_Toc391371861)

[1. Approaches based on additional structures 8](#_Toc391371862)

[2. Approaches based on heuristics 12](#_Toc391371863)

[IV. The solution 13](#_Toc391371864)

[V. The implementation of the solution 15](#_Toc391371865)

[V.1. DISCO 15](#_Toc391371866)

[V.2. DISCO Formalism 15](#_Toc391371867)

[V.3. Development environment 16](#_Toc391371868)

[V.4. Challenges of the implementation: 16](#_Toc391371869)

[VI. FUTURE WORK 17](#_Toc391371870)

[VII. References 17](#_Toc391371871)

# Introduction

Researchers in Artificial intelligence, and especially in the field of automated reasoning and problem resolving, are interested on the representation of the real world using logical models to define processing algorithms for these models. The reasoning about actions and changes is one of the fields in the IA which is particularly interested to some problems involving world changes. In particularly, since the 80s, Planning allowed to propose several automatic methods that, from an initial state, a goal state and a set of actions described as transitions between states, build a sequence of actions that lead from the initial state to the goal state.

Once the plan is built, it will be executed by the controller of the physical system. However, sometimes during the execution, the current state may not correspond to the expected state. Therefore, the controller can no more proceed with the plan. This is called a breakdown.

These breakdowns can be caused by dynamic environment (extern actions that modify the system states), or by an incomplete modeling of the real system. The planner needs, in order to build a consistent plan, a complete and a faithful representation of the problem actions: knowledge domain. Nevertheless, modeling such complete domain will require signiﬁcant knowledge-engineering effort if it is not impossible [9]. For example, if we want to represent the human activity in housing for the intelligent management of energy [11], it is impossible to define all the possible actions and devises that form the environment. In the real life, we frequently observe a combination of these two phenomena: an incomplete model and a dynamic environment. Therefore, it is necessary to define plan repair in order to face possible breakdowns. This represents the topic of my master thesis that I’ll present here.

The goal of our approach is to build a system that can recover from breakdowns taking into account the incompleteness of the model. Unlike the existing systems which suppose that the model is pretty complete to be consistently fixed and the breakdowns are only caused by the dynamic nature of the environment.

In the following, I will present the existing works in this domain. The section II presents the linear methods of planning, the hierarchical planning and the reactive approaches and discussing their advantages and limits. In the section III, I propose a hybrid model that combines the execution of a reactive HTN with classic planning models of plan reparation. In the fourth section, I present the proposed implementation of this model and the work to be done for the rest of my internship.

# Background

Planning is among the oldest fields of Artificial Intelligence: it was introduced in the 60s when researchers began considering automated reasoning about actions and changes. The goal of planning is, given a set of actions that describe transitions between states, to find a plan, i.e. a series of actions to go from an initial state to a specific goal. Such AI planners are used to solve real world problems by reasoning on a *model* of the world. It has many applications in Robotics [16], Logistics [18].

In this chapter we will formally deﬁne the problem we attempt to solve, discuss related works and introduce the proposed solution.

## II.1. Classical and linear planning

STRIPS [8] is the first major AI planner. It considers domain knowledge, i.e. a representation of the real world, as a triple: *D = (S, A, ϒ),* with *S* a set of states, *A* a set of actions and *ϒ: S x A → S*, a transition function that maps an initial state and action to the resulting state after performing the action. STRIPS assumes that the actions in the model are deterministic and discrete in time.

Each state *s Є S* is a conjunction of positive literals. The world is monotonous (i.e. non-represented literals are false in the state).

Each action *a Є A* is of the form *A (P, AL, DL)* with:

* *A* the action symbol composed by a set of terms (e.g. move(X,Y,Z) to represent the action that moves object X from location Y to location Z);
* *P* is the precondition of the action, i.e. a conjunction of positive literals that must be true before the execution of the action;
* *AD* and *DL* define respectively the add list and the delete list. The former is a set of positive literals added to the current state after the action execution and the later represents the positive literals removed from the state after the action execution.

A planning problem is defined as a triple *P = <D, si, sg>* with *D* the domain knowledge, *si Є S* the initial state and *sg Є S* the goal state. A solution to *P* is a plan π, i.e. a sequence of actions *a1,.., an Є An* such that *ϒ (Si, π) → Sg* (starting from the initial state, the plan leads to the goal state).

Several methods exist to solve a STRIPS planning problem. It has been proved [5] that solving a planning problem in STRIPS is PSPACE complete. Several heuristics have been developed to find solutions in reasonable time. The initial STRIPS mean-end planner relied on a simple backward chaining search in the state space. In the 90s, most research focused on searching the plan-space (e.g. [17]). More recent approach such as GRAPHPLAN [2] and O-plan [6] propose to convert the planning problem to a different structure (e.g. a graph) for better performance. STRIPS was the foundation of several later works such as PDDL[12], , etc and there still exists an active research field to find better (and faster) planning algorithms.

## II.2. HTN planning

To work properly, the planner requires that the STRIPS problem is provided with a complete model of the domain knowledge, i.e. that the STRIPS representation corresponds to the real world situation. This modeling requires signiﬁcant knowledge-engineering effort: one of the major difficulties with STRIPS is that all actions are at the same atomic level. As the considered problems become more and more complex, describing all these actions independently turns out to be an impossible task. This is the reason why new models have been proposed. Hierarchical Task Networks (HTNs) are probably the most commonly used model for planning since the 2000’s [15].

*Overview of HTN:*

A HTN or Hierarchical Task Network can be thought of AND/ OR tree structure (see Fig.1.1) where the root node expresses the task to be achieved. Tasks (T) and recipes (R) are alternatives nodes at each level in the tree.

Tasks nodes are OR nodes whose children are recipes that can be used to decompose the respective task into more primitive subtasks. Recipes nodes are AND nodes because the children of the recipe are a set of partially or fully ordered tasks resulting from using a recipe to decompose a task, all of which must be performed when applying the recipe to decompose a task.

Non-leaf node tasks represent compound tasks that need to be decomposed by a recipe and the leaf nodes are primitive tasks that can be directly executed.

**Each node has a set of zero or more modeling specification depending on the HTN modeling. These modeling specifications are pre-conditions, post-conditions and applicability conditions.**

* *Applicability conditions:* is the modeling specification of recipe’s nodes whichhelpsthe HTN choosing the appropriate decomposition if there is more than one.

**The task nodes haves two types of modeling** specifications**:**

* *Pre-conditions:* Specify when a task can be executed, so no task will start until all of its preconditions are satisfied.
* *Post-conditions:* Specify when a task execution success or fails, in fact a task execution is considered as complete when its post-conditions are satisfied.

**AND**

**OR**

**OR**

**AND**

**AND**

**OR**

**AND**

**AND**

R11

R21

R31

R32

R21

R22

R23

R1

**Goal**

T11

R2

T12

T13

T111

T112

T21

T2100000000

T112

T131

T321

T322

T212

T213

T222

T231

T212

**Figure 1.1:** HTN tree representation.

* **HTN formalism:**

We start the HTN formalism by the presentation of HTN compound namely tasks and recipes.

* ***Task formalism:***

**Tasks are expressions of the form *T (PC, EC)* whereT is a symbol defining the task name. *PC* is the pre-conditions and *PC* is post-condition modeled as set of literals.We denote two types of tasks; *compound task* or non-primitive task and *primitive task* or action. For example figure 1.2 represents a compound task** *Move (Obj, Room1, Room2, Door)* which is decomposed into three primitive tasks *{ Pick-up (Obj), Walk (Room1****,*** *Room2, Door), Put-down(Obj)}.*

**Move (Obj, Room1, Room2, Door)**

**Recipe 1**

**Walk (Room1, Room2, Door)**

**Put-down (Obj)**

**Pick-up (Obj)**

**Figure 1.2: HTN example representation.**

* ***HTN Recipe formalism:***

**An HTN recipe is used for the decomposition of compound task and has the form of a triple *R = (N, T, H)* where *N* is the name of the recipe which is unique such that no two recipes can have the same name. *T* is a non-primitive task performed with this recipe and *H* is a task network containing the set of subtasks of m to describe one way of performing the task *T*.**

**Using the HTN components formalism, we present now the domain knowledge and the HTN problem planning.**

* *The Domain* ***knowledge*** D is a list of all the recipes in the HTN needed to decompose the HTN tasks. *D= [R1,…, Rn].* Thus an HTN is a tree defined as a tuple *M = <T, D >* where *T* is the task network composed by all the tasks of the HTN and *D* is the domain knowledge.

*The planning problem* P is defined as a tree-tuple *P =<I, M, G>* where *I* is the initial state, *M* is the domain problem defined by the HTN tree and *G* is the goal task to achieve. G belongs to the tasks of M and the precondition of G is true in I. The problem is to find a plan that solves P. HTN planning works by expanding tasks and resolving conflicts iteratively, until a conflict-free plan can be found that consists only of primitive tasks.

Planning proceeds using *task decomposition* that starts from the initial goal task G, expands this task using a corresponding recipe R, and breaks down the goal into sequence of simpler sub-tasks. This process is applied recursively until the planner reaches a sequence of fully ordered primitive tasks that can make a goal successful. Thus, the solution of the planning problem P is a plan π = [*a1,.., an* ] is a sequence of primitive tasks ai *Є* M , and i Є [1,n].

The HTN planners are becoming popular in different domain where several systems were developed these recent years such as SHOP [13], SIPE [20] or NOAH [15].

General purpose of planning as STRIPS and HTN relies on a complete model of the domain knowledge that means if the model is not complete the planner won’t be able to plan. This approach of planning is named Declarative representation. Such representation of HTN requires a full **modeling of HTN domain knowledge. Each task in the HTN has at least one modeling specification that allows reasoning in planning. Despite the completeness of this representation, m**odeling such domain require signiﬁcant knowledge-engineering effort if it is not impossible**.**

## II.3. Reactive HTNs:

Declarative approaches planning attempt to predict the future. The planning is made in off-line phase that start from initial state to search for logical future states to reach a final goal state, using for that the modeling specifications in the domain knowledge. Once the plan is generated, the execution phase is launched in the environment.

In contrast with this approach, reactive HTNs aboard an approach with no planning phase that use a lazy approach called least commitment. Moreover, because the complexity of modeling a domain knowledge, reactive HTNs run through a hand-authored HTN with a procedural definition of the modeling specifications. Procedural definition means that all conditions can only be evaluated in the current state.

The solution of a planning problem is to find a path through the HTN using recipe selection. Starting from the initial state and the goal task, The HTN incrementally generates a sequence of sub-goals by choosing recipes for decomposing tasks. The recipe is selected by the applicability condition at the state where this last can be evaluated. The modeling specifications are evaluated in the current state, if it returns the values true the HTN consider it as valid and continue the exploration of the HTN. However if the evaluation of any specification fails (returns false), the HTN execution fails and we say that the HTN reaches a dead end. This situation is considered as hard failure. The execution is considered as complete if primitive tasks that make the goal successful are reached. Performing systems exist in the literature as the systems Disco [14] and RavenClaw [4].

## II.4. Plan execution, breakdown and recovery:

The declarative planners introduced before return a plan π composed of a sequence of primitive tasks (actions), which is passed to the controller for the execution phase.

The execution starts from the initial state and has to achieve the execution of all the actions of the plan to reach the goal state.

The execution of a STRIPS action is valid if its preconditions match subsets of the current observed world state immediately before the action is executed. The effects the executed action are added to the resulting state and the deleted effects are removed from it.

The HTN execution takes the STRIPS execution one step further by introducing the post-condition checks. As STRIPS primitive task (action) is applicable if its preconditions are valid in the current state of the world, in addition, the execution of the primitive task is considered as succeed only if its postconditions are part from the resulting state.

Thus, the plan is a solution to a planning problem if it matches a subset of the current world state immediately after the last action in the plan is executed.

As the environment is dynamic, the controller monitors the environment at each execution step in order to prevent a breakdown. A breakdown occurs if any deviation from the steps defined in the constructed plan is detected, such deviation makes the executed plan invalid and the execution of plan stuck.

The breakdown is identified if the world changes in an unexpected way that causes the execution failure of an action. The failure can be caused by one of these situations:

* First, the action preconditions are no longer satisfied in the current state. For example, a robot that plans to walk to room1 to room2 through door1, arriving at door1 the wind blows and the door1 became close, this unexpected change invalidate the precondition of the action that is the door must be open.
* Second, the executed action leads to unexpected consequences, in this case the postconditions of the action are not satisfied.

## II.5. the problem

Now considering reactive HTNs where the valuation of modeling specification is optimistic. This execution approach can reach a dead end. In the absence of planning process there is no prediction of the future step to follow, and the controller might choose a way in the HTN that lead to a certain task which is not applicable (applicability conditions are not valid in the current state) . In such case, we say that the HTN executor made a breakdown.

In such position where the HTN execution fails (face a breakdown). The controller has to find a different way in the HTN to reach the final goal state.

# Related work

The idea of reusing an existing plan instead of planning from the scratch is not a new idea. Thus, we found in the literature many works from several researches who were interested by developing planning systems that include plan repair and replanning capabilities to deal with any unexpected events while the plan is executed in a real-world environment. Most of these works were based on two approaches; the first is to build an additional structure with the produced plan to help the planner in the replanning phase. The second approach uses a planning heuristic to choose the most promising tasks to refine.

In this section, we discuss the most recent of these approaches.

## Approaches based on additional structures

The first system we look for is **Replan** system [3].Their formalism of HTN is an extension of the declarative planning HTN formalism.

The task architecture used in **Replan** is based on two hierarchies. On one hand, the sequential abstraction hierarchy that contains complex tasks. These tasks type subsume more primitives one. On the other hand, the abstraction hierarchy which is a task decomposition hierarchy such that a task type can be substituted with a sequence of tasks, this type of task is called abstract task. The definition of actions was also augmented, in fact the action’s effects are composed by effects of the action when the preconditions of the action hold and the effects of the action when its precondition do not hold.

In addition, each task in the Replan system has a utility interval which express the upper and lower bounds of the best and the worst outcomes by refining the defined task. The produced utility is used to choose the best decomposition when refining (decomposing) a task.

The planner starts the decomposition from the top goal task, and at each refinement step, the planner computes the expected utility of a partial plan generated by projecting it from the current world state, afterward a pruning heuristic is used to eliminate all the suboptimal plans that have their utilities dominated by some other plan. For the resulting plan, Replan constructs a derivation tree that describes how the plan was derived and includes all the tasks involved in the construction of the plan. This tree is essential for the plan recovery process.

During the execution process, a breakdown is identified in the case where the plan doesn’t reach the goal or that it reaches with a very low utility compared to what was expected in the planning process.

The replanning algorithm is based on a *partialization* *process.* The partialization process starts by finding the leaf node in the derivation tree whose preconditions are not hold in the current world. This action marked as *focused action (FA)*.

If the FA is subsumed by an abstract task, then the FA and the abstract task are removed from the derivation tree. However if the FA appears in the decomposition of a complex task, then the planner will search for a descendent of a sibling node of FA which has not been executed yet, if the action exists, this later will be marked as the current FA.

However, if all the siblings of the FA have been refined, then all the siblings and the FA are removed from the derivation tree, and the parent complex task become the current FA.

The partialization process stops when a promising plan is found, in the worst case the whole derivation tree is discarded. A promising plan is a partial plan that has higher bound utility than the old one. Otherwise the partial plan is discarded. The promising plan is refined to achieve the goal.

The next systems we present are built on the top of a famous HTN planner SHOP [13] and attempt to help the planner to recover from a plan failure. In The following we present these planning systems, but first we introduce briefly the SHOP planner.

SHOP is a HTN planning algorithm that is domain independent, which plans by recursively decompose abstract tasks until primitive actions are reached. In addition SHOP make a constraint that tasks must be planned in the same order that they will be later executed, thus the decomposition produced by each method has to be a total ordered set of subtasks. This constraint gives SHOP knowledge about the current state at each planning step which improves its degree of expressive power in its knowledge base. Nevertheless, as plans are computed in off-line phase, the world can change and causes an action fails.

In order to rectify this situation, two systems are proposed. These systems are built on the top of a modified version of SHOP. The first one is **HOTRiDE** [1], and the second is **RepairShop** [16]. In the following we introduce these two systems.

HOTRiDE is a planning system, (Hierarchical Ordered Task Deplaning in Dynamic Environments), which provides plan generation, execution, and plan repair system to recover from failure situations.

The planning process is based on SHOP planning algorithm, in addition to the resulting plan, HOTRiDe produces a task-dependency graph that consists of HTN traces generated by the planner that represents the dependencies between the tasks using causal links.

**Causal link (a, t):** a causal link between two task *t1* and *t2* is a pair (e, p). I

* In case where *t1* and *t2* are primitive tasks, then *e* is an effect of *t1,* and *p* is a precondition *t2.*
* If *t1 and t2* are compound tasks, then *e* is the effect of sub-action generated by decomposing *t1* and *p* is a precondition of *t2.* The causal link (e, p) means that the task *t1* supports the task *t2.*

**An HTN trace** for a task *t*consists of an ensemble of HTN trace nodes where each HTN trace node is defined as a tuple N= (t, π, A, D, Q, C) where t is a task, π is the plan that achieves the task t. A is a cumulative addition resulting from achieving the task *t* and D is the cumulative deletion made to the state while achieving the task t. Q represents the preconditions of t. and C is a set of pointers to the Childs nodes of the node N.

**A task-dependency graph** is represented as triple DG = (DT, CL, PL). Where:

* DT: HTN trace.
* CL: causal link list: for every p ground atom, CL (p) is an ordered list of heads of ground operator instances that add (delete) p to (from) the state.
* PL: is a totally ordered list of all the tasks that have a ground atom *p* as a precondition.

Once the task dependency graph is computed, the plan is passed to the controller for the execution process. As the environment is dynamic, the controller monitors the current state at each execution step, and tries to execute the action step in this current state. If at any point the controller detects a breakdown, HOTRiDE identifies the failed action i.e the action whose execution cannot end in the current state. Then HOTRiDE starts the recovery process.

The recovery process starts by first, checking every parent of the failed action using the task-dependency graph to identify the minimal failed parent.

The minimal failed parent is either a compound task that is identified as failed but whose parent is not, or a goal task that does not have any parents in the task-dependency graph.

For example, during the execution of a plan (fig 1), the execution of the action *D* failed, HOTRiDE traverse the hierarchy of the task-dependency graph in the upside-down manner to detect the minimal failed parent task of the action *D*.

Once the minimal failed task *A2* is detected, HOTRiDe detects the set of all causal links supported by the initial plan.

Figure 2 : Dependency graph

If the task *A2* is not the first descendent of its parent, then HOTRiDe marks only *A2* as minimal failed parent and invokes SHOP to generate a new decomposition for *A2* and its corresponding dependency graph.

Next, HOTRiDe establishes all of the causal links between any task of the new dependency graph and the tasks in the previous dependency graph that are not accomplished yet.

Nevertheless, if SHOP cannot find a new decomposition for *A2*, or all the decompositions found fail, HOTRiDE marks *A2* and its parent *A* as failed tasks, and attempts to replan for *A*.

After replanning, there may tasks in the original dependency graph that are no longer supported by the new plan. For example the task *B2* was supported by D in the original plan; if the task is no more supported then HOTRiDE marks it as failed task and attempts to generate other decompositions for it. If there is no such decomposition, HOTRiDE tries other possible decompositions for the minimal failed parent task *A2.*

This process is repeated for every causal link that is not supported in the new plan until SHOP generates a new plan that satisfied all the causal links. HOTRiDE executes the new plan starting from the failed action *D* in the current state of world. Otherwise HOTRiDe repeat the plan repair on the hierarchy of the parent of *D* until one of the following holds:

HOTRiDE generates a plan that is executed successfully in the world; or in the worst case the plan repair process marks a goal task as failed and replans from the scratch.

The repairSHOP[19] is similar than HOTRiDE in their approach, but differs in the structure of the tree that they use.

RepairSHOP augments the planning system SHOP with a direct goal graph (GG) to allow SHOP with replanning capabilities. The Goal Graph uses a Redux architecture that combines the theory of Justification-based Truth Maintenance System (JTMS) and Constrained Decision Revision (CDR). This combination provides the ability to the GG to perform dependency-directed backtracking in order to propagate changes when a failure occurs.

JTMS is dependency tree where nodes are called assertion. Each assertion is associated with a justification which is composed of two lists of assertions. The first is IN-list and the second is OUT-list. The assertions in the IN-list are connected via “+” links, while those in the OUT-list are connected by “-“links. The valuation of assertions depends on the validation of its justification.

A Justification is valid if every assertion in the IN-list is labeled IN and every assertion in the OUT-list is labeled OUT. A believable assertion is labeled IN and in contrary an assertion that cannot be believed is labeled “out”.

The GG is composed of goals, decisions and assignments. A goal might be performed by several recipes called decisions. The role of a decision is to decompose a goal into a set of sub goals, in addition each decision contains a task list that contains the decomposition of the performed goal and an assignment list that contains the conditions needed to perform the decision on a goal. The goal is represented by tasks in SHOP and assignments are the preconditions and the order constraints of tasks.

The GG is computed automatically during the planning process as follows:

* Each task is encapsulated in a goal structure in the GG
* Each primitive task successfully planned is added to its parent goal list
* The planner considers each possible decomposition for each compound task and encapsulates this reduction with its conditions in a new decision. If the task decomposition succeeds then the plan is added to the decision’s goal list and the GG marks the additional decompositions left not evaluated as NULL. These decisions are used later in the plan recovery process. If the task evaluation fails then the decision is marked as OUT.
* RepairSHOP constructs justifications for branches where a failure may occur.

The controller starts the execution process and attempts to decompose each compound task G using its corresponding decision O. When a decision O fails to decompose a task Ti, the planner labels the decision O as invalid. Then the plan repair process starts.

The controller check to see if alternate decision OG previously marked NULL is available to decompose the goal Ti this decision is labeled in the GG as valid decision. The GG is updated to records the modifications. Otherwise if any decision is found to decompose Ti then the controller backtracks in the GG to propagate the result to the highest affected goal and returns the first available alternate decision from the nearest goal node. If an alternate decision is found, the controller calls SHOP to replan from this goal.

Systems based on additional structures demonstrate theirs efficiency. Nevertheless, one of the limitations of this approach is that the plan repair is depended to the initial planner. If the latter is unable to find new decompositions for the failed tasks, then the repair plan algorithm will return any solution and the plan execution will fail. Therefore, a heuristic based approach was introduced.

## Approaches based on heuristics

The system presented is an HTN forward chaining planner combined with an A\* like heuristic search, to plan, execute and adapt a robot plans in dynamic environment. This system is called Dynagent [10].

The planning algorithm proposed in this system is forward HTN planner similar to the SHOP planner presented previously. In addition, this system extends the notion of the HTN’s components for the purpose of replanning. These compounds are called tasks plus, actions plus and plan plus.

For each task plus preconditions were extended to protected conditions and remaining conditions. The satisfiability of the protected conditions have been confirmed in the process of planning, and they will be used to detect the invalid plans when the belief is updated. The satisfiability of the remaining conditions remains to satisfy during the planning process. An action plus is considered as solved action plus if its remaining set is empty.

In addition, each action plus records the initiation set and the termination set. The initiation (respectively, termination) set records the primitive fluents which start (respectively, cease) to hold after the action execution HTN planning [10].

The plan plus records two types of plans, solved plan plus which all actions that it contains are solved actions plus and supplementary plan plus is a plan which contains a task plus or action plus such that one of the fluent in its remaining set is marked as invalid. The supplementary plans are later used to generate new valid plans when the fluent becomes valid.

The recipes used to refine tasks are HTN rule for decomposing abstract task and actions rule that are used to process primitive actions or tasks. These recipes contain the preconditions that must be satisfied before refining a task plus and action rules. Each task has an estimated cost from refine it and the cost of a plan is the sum of each cost of task in this plan.

The system introduced in this section proves their efficiency in their respective domain of application. However, we consider that the proposal do not offer a flexible approach to tackle the incomplete definition of the domain knowledge of reactive HTN. The additional structures are constructed using the domain knowledge of planner. Without this knowledge the dependencies trees are incompletes and invalid during replanning.

The agent starts observe the current world at each stat and update the belief as follows:

* If a fluent is deleted then the planner deletes all the invalid plans from the current set of plans and the set of supplementary plans.
* Otherwise, if a fluent is added, the planner make the new valid plans from the set of supplementary plans

The planning process starts by detecting all the applicable plans plus. Next an A\* heuristic is called to choose the best plan plus to refine based on the cost of the actions composing the plan.

The planner proceeds to the refinement of the plan plus by decomposing the tasks in this plan and replace its occurrences in the current set of plans plus. This procedure is repeated until each remaining conditions of tasks involved in the current set of plans plus become empty. The planner returns a solved plan plus and update the current set of plans plus and the set supplementary plans.

During the execution the belief is continuously updated to monitors the execution of actions in the plan.

Thus, when an action execution is successful it executes the following procedure

* Delete from the current set of supplementary plans plus all the plans plus whose first element is not a solved action plus.
* The fluents in initiation set of the executed action are added to the current belief and the fluents in the termination.
* After the belief update, the planner deletes all the invalid plans plus from the current set of plans plus and the current set of supplementary plans.

Otherwise, when an action execution fails then the controller deletes each plans form the current set of plans plus and the current set of supplementary plans that contains an action which is unifiable with the failed action.

We have presented an agent system combining forward chaining HTN planning, A\*-like heuristic search. This system uses A\*-like heuristics for selecting the task to be decomposed, which is different from our solution that aim is to define a classical planner that can be used to recover from the breakdown.

The plan repair system presented above are based on partial replanning from the initial domain. They assume that modeling a complete domain knowledge is possible. Nevertheless, if the domain knowledge is always incomplete, then the plan repair (based on a dependency graph itself probably incomplete, or using a heuristic on an incomplete domain) will very likely lead to a new breakdown. The system will therefore never reach the goal state. In addition the use of a heuristic requires a declarative formalism of tasks to evaluate them, which makes this approach inapplicable in the case of a reactive HTN.

# The solution

In this section, we introduce our solution for plan recovery in reactive HTN. We also present the algorithm of the proposed solution.

Reactive HTNs are based on procedural definition of the domain knowledge. This definition doesn’t contain any logical information to enable the planner reasoning about task decomposition and execution. Thus, if the HTN faces a breakdown during the execution, it will be unable to backtrack in order to find another decomposition that achieves the execution of the task.

In order to face this problem, we propose the construction of a hybrid system that integrates to a reactive HTN a reasoning engine modeled by a STRIPS planner. The proposed system will be able to generate a linear plan to repair the failed task starting from the current state in the world and only based on partial information available in the domain knowledge of the HTN. In addition, we propose to extend the definition of certain tasks in the HTN from procedural definition to a declarative definition in order to allow the STRIPS planner to reason. An overview of the proposed execution and plan repair system is illustrated in Figure 3.

Controller

Observe

Execute(Action)

Succes

failure



**Current State**

Execute(Action)

**HTN**

**Hybrid system**

Reactive DC

**Domain knowledge**

**Declarative DC**

**STRIPS**



**Breakdown**

**generate (Plan)**

Figure 3: High-level description of the system architecture.

The process of plan repair starts where the HTN detects a breakdown, this later has to communicate to STRIPS the failed task and call the repair algorithm (STRIPS planner).

First, STRIPS has to constitute the planning problem to reason about:

* STRIPS construct its domain knowledge by extracting partial information from the HTN domain knowledge and extends them to declarative definition.
* The goal state defines one of the failed task conditions (preconditions or postconditions) the system attempts to replan.
* The current observable state is defined as the initial state.

Next, STRIPS try to generate a plan to repair the failed task, if any solution is found based on the partial domain knowledge then STRIPS will search for another conditions to satisfy by searching in the HTN a parent task in the hierarchy or a sibling task to the failed task. The produced plan is then passed to the controller to be executed in the current state. Finally, the controller will ask the HTN the next task to execute starting from the current state.

To illustrate our solution, considering the simple HTN in the figure 1.1 that describe a robot which plans to move an object from room1 to room2, through door1. In the execution process the robot is in the room1 starts by picking up the object and walk toward room2. However, arrived at the door1, the robot finds that the door is not open because the wind blows and the door is now closed. The HTN fails because the preconditions of the action *walk to room2 (IsOpen (Door))* are no longer valid in the current state. The agent has to change its plan and find another sub-plan that leads him to the room2 starting from the observable current state. The controller will ask STRIPS a plan that task as goal the state where *IsOpen (Door)* is valid. Using the partial domain knowledge constructed, STRIPS will generate a plan of two actions {Unlock(Door), Open (Door)} as it is shown in figure 4.

*isOpen(Door)*

**Move (Obj, R1, R2, Door)**

Walk (Room1, Room2, Door)

Put-down (Obj)

Pick-up (Obj)

**Move&Paint (Obj)**

**Paint(Obj)**

**….**

**…**

Figure 4: HTN pour peindre un objet

**RepairPlan= { Unlock(Door), Open(Door)}**

# The implementation of the solution

In order to allow the implementation of our solution, we used the dialog system DISCO [12] as reactive HTN to which we integrate a STRIPS planner to support the plan repair process.

## V.1. DISCO

Disco is a system based on the reactive architecture. its functional architecture is based on two mains features:

* A reactive formalism that allows the system to lead the user in a real time, without making any plan in advance. The task decomposition is managed by two components, a generic task engine to validate the tasks model and maintaining a representation of the current status.
* A user interface to communicate with the user in the case where the engine needs more information or the user help to achieve certain task.

## V.2. DISCO Formalism

Disco is implemented on ANSI/CEA-2018 standard which uses JavaScript for the procedural definition of conditions (PC, EC and AC) in the domain knowledge. The formalism includes the tasks modeling, input, output, preconditions and post conditions.

The tasks are modeled in XML format. In addition, the tasks composition has been extended in order to support the following components in the runtime:

* Input including all data that affect the execution of the task.
* Output including all data that can be affected by the execution of the task
* Primitive tasks may contain SCRIPT parameter which represents a JavaScript program that is executed in the environment as a condition.

An example of DISCO formalism is presented in figure 3.

## V.3. Development environment

Considering the incompatibility of the two environments (Java/Prolog), we decided to work with a hybrid platform TuProlog that allows the integration of the STRIPS planner which is implemented in Prolog into the JAVA environment.

This platform is mainly used as a communication bridge between these two environments and supports the conversion from the reactive formalism to the declarative formalism.

<taskModel ...>

<task id="MoveAndPaint">

<subtasks ...>

<step name="move" task="Move"/>

<step name="paint" task="Paint"/>

...

</subtasks>

</task>

...

<task id="Walk">

<input name="door" .../>

<precondition> isOpen($door) </precondition>

...

</task>

...

<task id="Paint>

<subtasks ...>

...

</subtasks>

</task>

...

</taskModel>

Figure 3: Domain knowledge Formalism ANSI/CEA-2018.

## V.4. Challenges of the implementation:

The most important challenge within the realization of a hybrid planner is to support heterogeneous formalisms used to model the system. The system must be able to convert the reactive HTN domain knowledge to declarative domain knowledge capable to run in Prolog and to handle exceptions related to this conversion.

In addition, the incompleteness of domain knowledge requires a reflection on the level of information necessary to introduce in STRIPS domain knowledge to allow this later to reason (plan).

Finally, designing a planning algorithm able to provide effective solutions based on incomplete information in dynamic environment.

# FUTURE WORK

My internship begun on mars 26th for a period of six months. The first part of this internship was devoted to study the problem and all the current state of art.

In fact, during the first month, I studied the reparation plan methods in HTNs in order to situate the problem. The second month (May) was devoted to study the reactive HTNs problem and the writing of the first part of my work.

Since the beginning of June, I’m working on the conception of a planning algorithm that combines a STRIPS planner and DISCO.

The rest of this internship will be devoted firstly to the implementation of the local reparation algorithm (that we presented in the third section). Next, I will test and validate this algorithm with real problems especially for the interaction human-machine. I’ll map the working assumptions of our plans repair algorithm, and from the constraints, I will develop a modeling support system that propose to the modeler of the HTN to add some declarative information in the HTN in the parts where a breakdown may occur. This system is used to prevent breakdown.

# References

1. Ayan, N. Fazil, et al. "HOTRiDE: Hierarchical ordered task replanning in dynamic environments." Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop. Providence, RI. 2007.
2. Blum, Avrim L., and Merrick L. Furst. "Fast planning through planning graph analysis." Artificial intelligence 90.1 (1997): 281-300.
3. Boella, Guido, and Rossana Damiano. "A replanning algorithm for a reactive agent architecture." Artificial Intelligence: Methodology, Systems, and Applications. Springer Berlin Heidelberg, 2002. 183-192. 2002.
4. Bohus, Dan, and Alexander I. Rudnicky. "RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda." (2003).
5. Bylander, Tom. "The computational complexity of propositional STRIPS planning." Artificial Intelligence 69.1 (1994): 165-204.
6. Currie, Ken, and Austin Tate. "O-Plan: the open planning architecture." Artificial Intelligence 52.1 (1991): 49-86.
7. Erol, K., Hendler, J., & Nau, D. S. (1994, October). HTN planning: Complexity and expressivity. In AAAI (Vol. 94, pp. 1123-1128).
8. Fikes, Richard E., and Nils J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." Artificial intelligence 2.3 (1972): 189-208.
9. Gil, Yolanda. Acquiring domain knowledge for planning by experimentation. No. CMU-CS-92-175. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1992.
10. Hayashi, Hisashi, et al. "Dynagent: An incremental forward-chaining HTN planning agent in dynamic domains." Declarative Agent Languages and Technologies III. Springer Berlin Heidelberg, 2006. 171-187.
11. Huraux, Thomas, et al. "Un modèle multi-niveau pour simuler l’activité humaine dans le contexte de la consommation énergétique résidentielle."
12. McDermott, Drew, et al. "PDDL-the planning domain definition language." (1998).
13. Nau, Dana, et al. "SHOP: Simple hierarchical ordered planner." Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2. Morgan Kaufmann Publishers Inc., 1999.
14. Rich, Charles. "Building task-based user interfaces with ANSI/CEA-2018." Computer 42.8 (2009): 20-27.
15. Sacerdoti, Earl D. A structure for plans and behavior. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1975.
16. Shin, K. G., & McKay, N. D. (1986). A dynamic programming approach to trajectory planning of robotic manipulators. Automatic Control, IEEE Transactions on, 31(6), 491-500.
17. S. Russel, & Norvig, P. (2004). Inteligência artificial. Editora Campus.
18. Ricciardi, Crainic, T. G., N., & Storchi, G. (2009). Models for evaluating and planning city logistics systems. Transportation science, 43(4), 432-454.
19. Warfield, Ian, et al. "Adaptation of Hierarchical Task Network Plans." FLAIRS Conference. 2007.
20. Wilkins, David E. Practical planning: Extending the classical AI planning paradigm. Morgan Kaufmann Publishers Inc., 1988.