Pictures/
urlcolor=blue, colorlinks=true

1.3
i
pdftitle= pdfsubject= pdfauthor= pdfkeywords=

Doctoral Thesis

Author:
http://www.johnsmith.com

Supervisor:
http://www.jamessmith.com

A thesi submitted in fulfilment of the requirements
for the degree of

in the

September 1, 2014

Contents

CONTENTS

LIST OF FIGURES

List of Tables

LIST OF TABLES

1. INTRODUCTION

Chapter 1 . *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 [**?**]. For example, if we want to represent the human activity in housing for the intelligent management of energy [**?**], such as cooking dishes task. The most challenging part in modeling this task is to define the level of granularity with which we can construct a model that represent accurately the real world which implies representing each action of cooking and devices taking into account the variability of the environment. constructing such model is so time consuming even impossible. Thus, the existing models represent the general view of the world as showed in the figure **??** that represent the cooking example.

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 we will 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, we will present the existing works in this domain. The chapter 2 presents the linear methods of planning, the hierarchical planning and the reactive approaches and discussing their advantages and limits. In the chapter 3, 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, the experiments and validation of the proposed solution . We conclude this thesis by the future works.

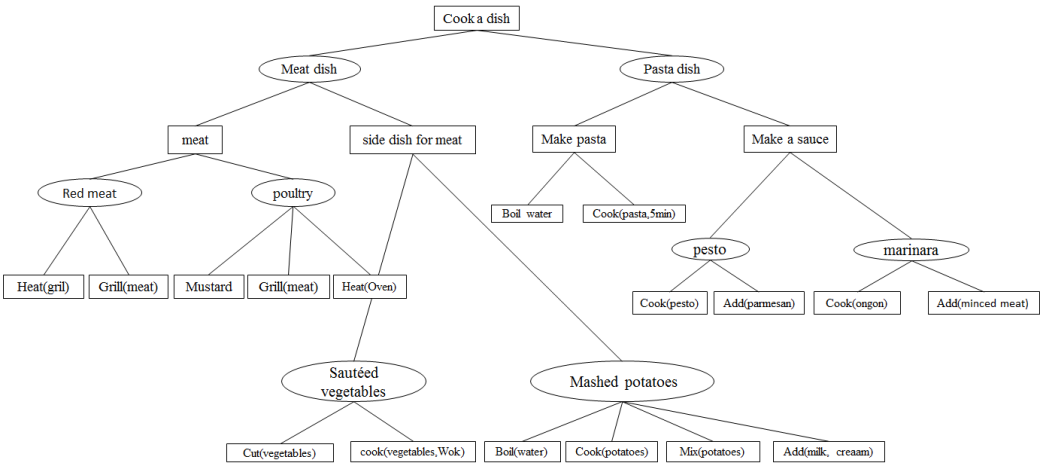

Fig. 1.1: Cooking model example

2. BACKGROUND

Chapter 2. *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 [**?**], Logistics [**?**]. ect.

2.1 Classical and linear planning

STRIPS [**?**] is the first major AI planner. It considers domain knowledge, i.e. a representation of the real world, as a triple: $D = (S, A, \gamma)$, with S a set of states, A a set of actions and $\gamma: S \times A \rightarrow S$, 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 $s \in S$ is a conjunction of positive literals. The world is monotonous (i.e. non-represented literals are false in the state). Each action $a \in A$ 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, $si \in S$ the initial state and $sg \in S$ the goal state. A solution to P is a plan π, i.e. a sequence of actions a1,.., an $\in$ An such that γ *(Si,* π*)*$\rightarrow$ *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 [**?**] 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. [**?**]). More recent approach such as GRAPHPLAN[**?**] and O-plan [**?**] 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[**?**], , etc and there still exists an active research field to find better (and faster) planning algorithms.

2.2 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 2000s

2.2.1 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 recipes nodes which helps the HTN choosing the appropriate decomposition if there is more than one.

The task nodes haves two types of modeling specifications:

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

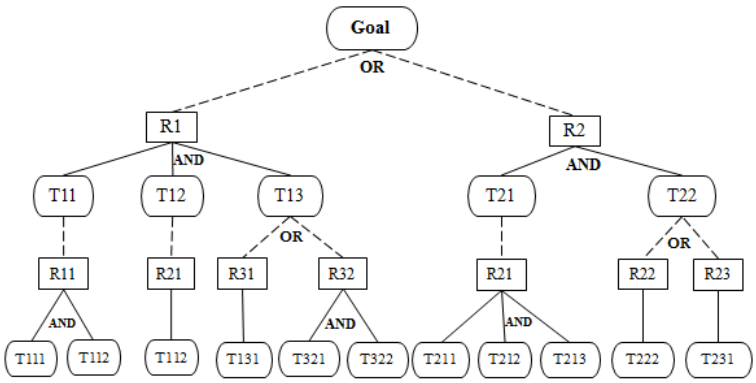

Fig. 2.1: HTN tree representation

2.2.2 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) where T is a symbol defining the task name. PC is the preconditions and PC is postcondition 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 **??** 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)*}.

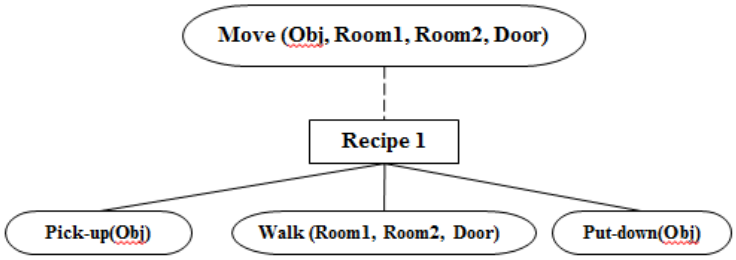

Fig. 2.2: HTN example representation

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 T describing 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 subtasks. 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 $\pi = [a1,.., an\ ]$ is a sequence of primitive tasks $ai \in M$, and $i \in [1,n]$.

The HTN planners are becoming popular in different domains where several systems were developed these recent years such as SHOP[**?**], SIPE [**?**] or NOAH [**?**].

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 wont 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, modeling such domain require significant knowledge-engineering effort if it is not impossible.

2.3 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 to predict the future states; instead, they compute just the next act in every instant. Moreover, because of 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. Reactive HTN are becoming very popular in the field of controlling complex artificial intelligent systems exist performing in dynamic environment such as Dialog with the Disco system [**?**] and RavenClaw [**?**].

2.4 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 postcondition 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 stops. 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.

2.5 The problem

Reactive HTNs take the position of using no planning process to detects the future state, instead, the HTN plan only for the next task to be executed and based only on the current observable state. In addition, the valuation of modeling specification is optimistic. This execution approach can lead the HTN to a state where no decomposition or execution are possible, this state is then called dead end. In such case, we say that the HTN executor faces a breakdown.Because the procedural definition of the domain knowledge and the reactive formalism of the HTN, this later is unable to backtrack in order to find another way to achieve the goal task. Therefore, the HTN can no longer continue its execution and the goal is impossible to achieve.

3. RELATED WORK

Chapter 3. *Related work*

3.1 Introduction

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 chapter, we discuss the most recent of these approaches.

3.2 Approaches based on additional structures

The first system presented is *Replan* system [**?**].The HTN formalism used for this system 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 actions 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 descendant 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 then refined to achieve the goal.

The next systems we present are built on the top of a famous HTN planner *SHOP*[**?**] and attempt to help the planner recovering from a plan failure. In The following we present these planning systems, after a brief introduction to SHOP planner.

SHOP is an HTN domain independent planning algorithm , which 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* [**?**], and the second is *RepairShop* [**?**]. 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 breakdown 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 representing 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).

- 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 to the current state 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 child 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 see figure.**??**, 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 in the example A2 is detected, *HOTRiDe* detects the set of all causal links supported by the initial plan. If the task A2 is not the first descendant 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

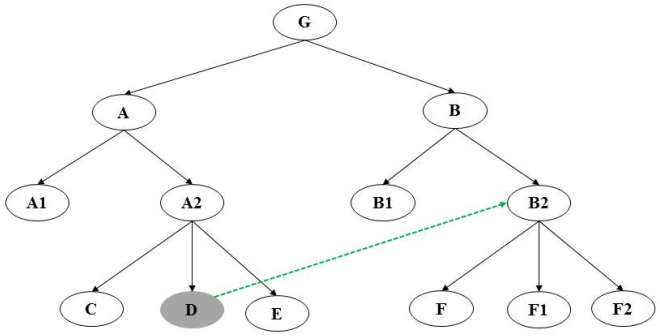

Fig. 3.1: Dependency graph example

case the plan repair process marks a goal task as failed and replans from the scratch.

The *repairSHOP* [**?**] is similar to 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 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
- 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 decisions 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.

3.3 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* [**?**].

The planning algorithm proposed in this system is a forward HTN planner similar to the SHOP planner presented previously. In addition, this system extends the notion of the HTNs 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 [**?**].

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 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.

The plan repair systems presented above are based on partial replanning from the initial domain and prove their efficiency on different real planning problems. Nevertheless, these systems present certain limitations. First, the plan repair approach used in these systems is depended to the initial planner. If the latter is unable to find new decompositions for the failed tasks, then the plan repair algorithm will return no solution and the plan execution will fail. Second, these systems assume that modeling a complete domain knowledge is possible, but as 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. Thus, we consider that the proposal do not offer a flexible approach to tackle the incomplete definition of the domain knowledge of reactive HTN.

4. DISCOLOG APPROACH

In this Chapter, we introduce the Discolog system, a reactive HTN planning, execution, and plan-repairing system. Discolog is a hybrid system that integrates a reasoning engine modeled by STRIPS planner to a reactive HTN. We will first present an overview of the system, and next we will detail the system architecture. Chapter 4. *Discolog approach*

Discolog uses reactive HTN style to achieve a goal with no prediction of future states at all. Starting from the top level goal, Discolog recursively decomposes tasks until it reaches a set of primitives tasks that can be directelly executed in the real world.

Nevertheless, because of procedural definition of reactive HTN domain knowledge which doesn't contain any logical information allowing the planner reasoning about task decomposition and execution, if the HTN faces breakdowns during the execution, it will be unable to backtrack finding another decomposition that achieves the execution of the task.

In order to face this problem, Discolog uses a STRIPS planner to propose a plan recovery. It starts from the current observable state of the world and uses only the partial information available in the domain knowledge of the HTN to reason about. For that matter, Discolog extends the definition of certain HTN tasks in the domain knowledge from procedural definition to a declarative definition . An overview of the proposed execution and plan repair system is illustrated in Figure **??** .

The plan recovery in Discolog comprises two main procedures. The first one attempts to detect goal candidates to repair and the second invoke the STRIPS planner to propose a plan repair for these candidates. these procedures are described in the next sections.

4.1 The Discolog algorithm

Let *HTN* be a model of an Hierarchical Task Network with a top level task *Goal* to achieve, and Disco the the reactive HTN .To achieve *Goal*, Disco

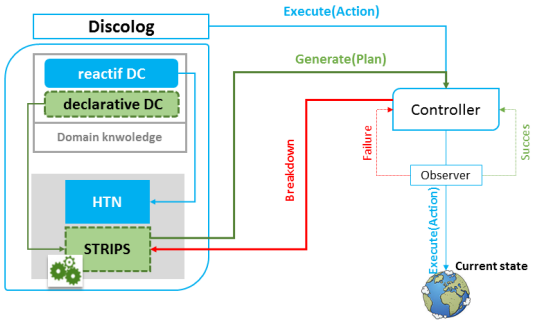

Fig. 4.1: High level description of Discolog system

proceed as follow: As showed in **??**, Discolog starts from the top level goal *Goal* and recursively decomposes tasks until it reaches a set of primitives tasks that can achieve *Goal*. Each task in Disco is defined with a *status(Task)* $\in$ {Live,Blocked,Done,Failed,Succeed}.

Before decomposing a non primitive task or executing primitives task, Disco evaluate the precondition of the this task. If the current state holds the preconditions(Task) then status(Task) is updated to Live. otherwise, Status(Task)= not Live and the HTN execution is blocked. The same, after the execution of a primitive task, Disco evaluate its postconditions. If post-conditions(Task) are valid in the current state the status(Task) is updated to done or succeed, otherwise status(Task)= failed and the HTN execution become blocked.

At the end of the process, Disco(HTN,Goal) returns either Success(Goal is achieved) or failure if Disco faces breakdown. These breakdowns are detected if the top level goal is not achieved i.e Status(Goal) != Done and Disco has no decomposition or execution to propose.

When such breakdown occurs, Discolog starts the recover procedure which will first look over the *Goal* and its children to find task candidates which can be repaired from the current state in order to recover from the breakdown. The recover procedure is defined in the next sections.

Algorithm 1 DiscoLog algorithm

```
procedure DISCOLOG(HTN,Goal)
    HTN ← ConstructModel()
    π ← Disco(HTN,Goal)
    if π ← Success then
        return Success
    else
        plan ← Recover(Goal)
        if (plan = null) then
            return Failure
        else
            for each action ai ∈ plan do
                Discolog(HTN,ai)

EndProcedure

procedure RECOVER(Goal)
    listCandidates ← findCandidate(G)
    if listCandidates = ∅ then
        return null
    else
        Π ← ∅
        for each candidate ∈ listCandidates do
            Π+ = InvokeSTRIPS(candidate, CurrentState)
            Cost ← {cost(π)|π ∈ Π}
    return π ∈ Π with minimum cost(π)

EndProcedure

procedure FINDCANDIDATE(Goal)
    for each child ∈ Goal do
        if (precondition(child)!=∅ and
            status(child) ∉ {Done, Live, Blocked}) then
            add precondition(child) to candidates
        else if (postcondition(child)!=∅ and status(child) ∈ {Failed}
then
            add postcondition(child) to candidates
        if (status ∈ {Live} and nonprimitive(child) and applicability(child)!=∅)
then
            add Applicabilitcondition(child) to candidates
        findCandidate(children(child))
    return candidates

EndProcedure
```

4.1.1 Goal candidates detection

When a breakdown occurs, Discolog will first look over the goal task and its children to determine tasks in the HTN which are compromised by the breakdown, theses tasks are defined as task candidates for the plan recovery. For each task, we extract the condition failed because of the breakdown:

- If the status of task is neither done nor live then the algorithm will attempts to repair its preconditions
- If the status of task is failed then its postcondition are not valid and the repair algorithm will attempts to repair these postconditions.
- if the task is nonprimitive and all its applicability conditions are invalid in the current state then the algorithm will attempts te replan to satisfy one of its applicability condition.

Once, the list of candidate is identified, it is then passed to the InvokeStrips procedure. As presented previously, a breakdown occur in a HTN if one of the task conditions fails, thus repairing a task using STRIPS planner is considered as repairing the failed condition of this task. Once, the list of candidate is identified, the prolog STRIPS planner is called to propose a recovery plan.

4.1.2 STRIPS repair planner

In order to generate a plan, STRIPS has to constitute the planning problem to reason about. First STRIPS constructs its domain knowledge by extracting partial information from the HTN domain knowledge and extends them to declarative definition. For that, Discolog convert all the primitive tasks to a declarative and logical formalism supported by STRIPS. Then for each candidate task, STRIPS takes as goal state the failed task condition of the task candidate and the current observable state is defined as the initial state.

Next, STRIPS tries to generate a linear plan to failed condition to reach a state where the failed condition is valid. Finally, Discolog calculate the best. i.e the plan the plan with the minimum cost of execution and convert its actions to reactive formalism. this plan is then passed to Disco to be executed in the environment.

4.2 Example

Lets the HTN describing the move&paint task execution presented in figure **??**. The HTN starts the decomposition from the top level goal and at each step it monitors the conditions of each task. Once it attempts primitive tasks, the execution in the real world starts. the HTN starts by executing the pickup(Object) by first evaluating its preconditions, after the execution of the task, the HTN evaluated the postconditions of the task. Arriving to the execution of the Walk task, , the world suddenly changes; assume that the wind blows and the door is now blocked thus the preconditions of the Walk(room1, room2, door1) task are no longer valid. Then a breakdown is detected and the recover procedure is called.

To recover from this breakdown, Discolog will first define the task candidates that can be repaired, which are the walk task with its preconditions and the move task with its postconditions.

Once the list defined, it is passed to the STRIPS planner, which creates its domain knowledge as shown in figure **??**, next it generates a plan for both candidates. STRIPS returns the best plan that can repair the walk task composed. In this example the best plan consists of two primitives tasks; unlock(door) and open(door). This plan is converted to Disco tasks and executed. Once the plan recovery executed, Discolog continue the task decomposition from the current state to achieve the moveandpaint task.

We present in this chapter the proposed solution and we detailed the conceptional procedure of this later. In the next chapter, we will present the implementation of the Discolog system.

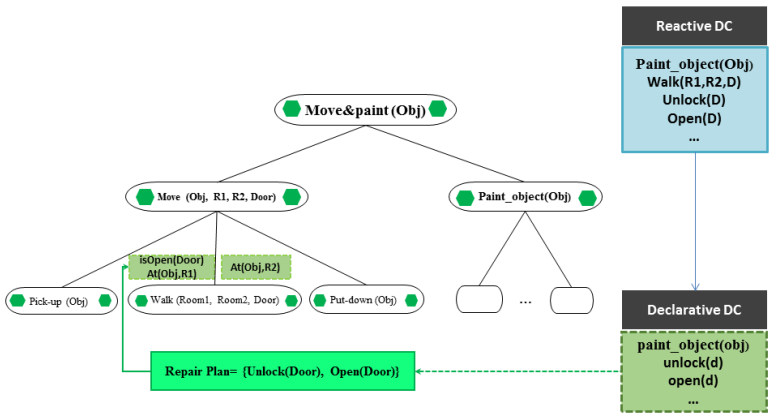

Fig. 4.2: Plan recovery for the move&paint task using Discolog

5. IMPLEMENTATION

Chapter 5. *Implementation* We described in previous chapters of the present thesis, the problem being addressed and our proposition for a solution. In the following, we present the implementation of this solution.

5.1 Development environment

In order to implement the hybrid system Discolog, we used the User Interface DISCO [**?**] as reactive HTN to which we integrate a prolog STRIPS planner to support the plan repair process.

The most important challenge within the realization of such hybrid system is to support heterogeneous formalisms of Disco ans STRIPS. The system must be able to convert the HTN domain knowledge from procedural formalizm implemented in Java to a declarative one capable to run in Prolog and handle exceptions related to this conversion.

for that purpose, we integrated to DISCO the tuProlog java-based lightweight Prolog engine to create a bridge that can use the logical prolog planner from the Java procedural environment . the environment architecture is described in figure **??**.

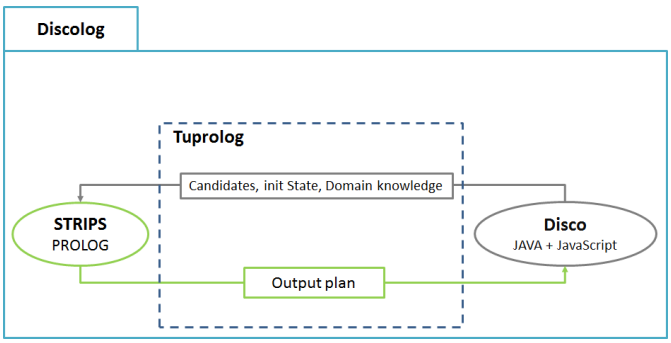

Fig. 5.1: implementation environment of Discolog

5.2 DISCO

DISCO [**?**]is a task based user interface with a reactive architecture. The most important feature of this reactive architecture is that allows the system to lead the user in a real time, without making any plan in advance. DISCO's functional architecture is composed by two mains components:

- A *task engine* whose function is to load and validate a task model description, and to maintain a representation of the current status of the users tasks.[**?**] In this present thesis, we only focus on this component.
- *User interface* to ensure the communication between the task engine and the user in the case where the engine needs more information or to help the user achieving certain task.

5.2.1 DISCO task model

Disco uses the ANSI/CEA-2018 standard for the procedural definition of the task model elements as described bellow :

- *Task*: The task model defines Task classes which are modeled using XML format.The figure **??** describes eight task classes, including three compound tasks and five primitive tasks. Primitive tasks may contain *grounding script* parameter defined as JavaScript program which represent the effect of the primitive task execution in the environment.
- *Inputs and outputs* : Input includes all the data that may affect the execution of the task, and output includes all data that can be affected by the execution of the task. these data type is defined in JavaScript.
- *Conditions* : Task's conditions (Preconditions, postconditions ans applicability conditions) are defined as boolean JavaScript function to evaluate the execution of a task.

5.3 Discolog implementation

For the implementation of the Discolog system, we faced some challenges. In addition to the management of heterogeneous environment, defining the level of information necessary to introduce in STRIPS domain knowledge required some reflection and designing a STRIPS planning algorithm able to provide effective solutions based on incomplete information in dynamic environment.

Fig. 5.2: Complete ANSI/CEA-2018 task model description for the moveandpaint task

5.3.1 The new Discolog API

In order to implement the hybrid system Discolog we create the API shown in figure **??**api divided on two mains folders :

Reactive DISCO

We create an extension of DISCO that can detect breakdowns and in such case, collects a candidate list to the plan recovery process.In addition we manage to generate these candidates in the easiest way to be converted into PROLOG formalism. The input PROLOG planner is constituted in DISCO via TUprolog as demonstrated in the procedure input.

```
private static void Strips_Input(List<String> Initial_state,
    String Goal,Prolog engine) {
        try {
        for(String init : Initial_state){
                engine.addTheory(new Theory("strips_holds(" + init +
                    ",init)."));
        }
        Theory PlannerCall = new Theory("test1(Plan):-
            strips_solve(["+ Goal + "],30,Plan).");
        engine.addTheory(PlannerCall);
        } catch (InvalidTheoryException e) {
```

```
        e.printStackTrace();
        }

}
```

Once the plan recover is generated, each action of this plan must be converted into primitive task formalism supported by DISCO. Therefore we create a procedure that treat and convert the STRIPS plan output as shown in the procedure output.

```
private static ArrayList<String> getPlannerOutput(Term plan) {
        ArrayList<String> Output = new ArrayList<String>();
        String init;
        if(plan == null)
                System.out.println("No recovery STRIPS plan found
                    !");
        else{
        Pattern p = Pattern.compile("(do\\()");
        String[] splitString = (p.split(plan.toString()));
        for (String element : splitString) {
                String elem = element.replaceAll("(init)(\\)+)", "");
                elem = elem.replaceAll(",", "");
                init = elem.replaceAll("\\(.*\\)", "");
                Output.add(init);
        }
        Collections.reverse(Output);
        }
        return Output;
}
```

Declarative PROLOG

The main challenge faced in the creation of the STRIPS planner was to create an efficient means-end planning algorithm that can be integrated and executed into TUprolog. Moreover, we had to define adequate structures that can receive the extracted domain knowledge from the DISCO model using Tuprolog.Thus, we extract all the primitives tasks and turn them to a PROLOG formalism.

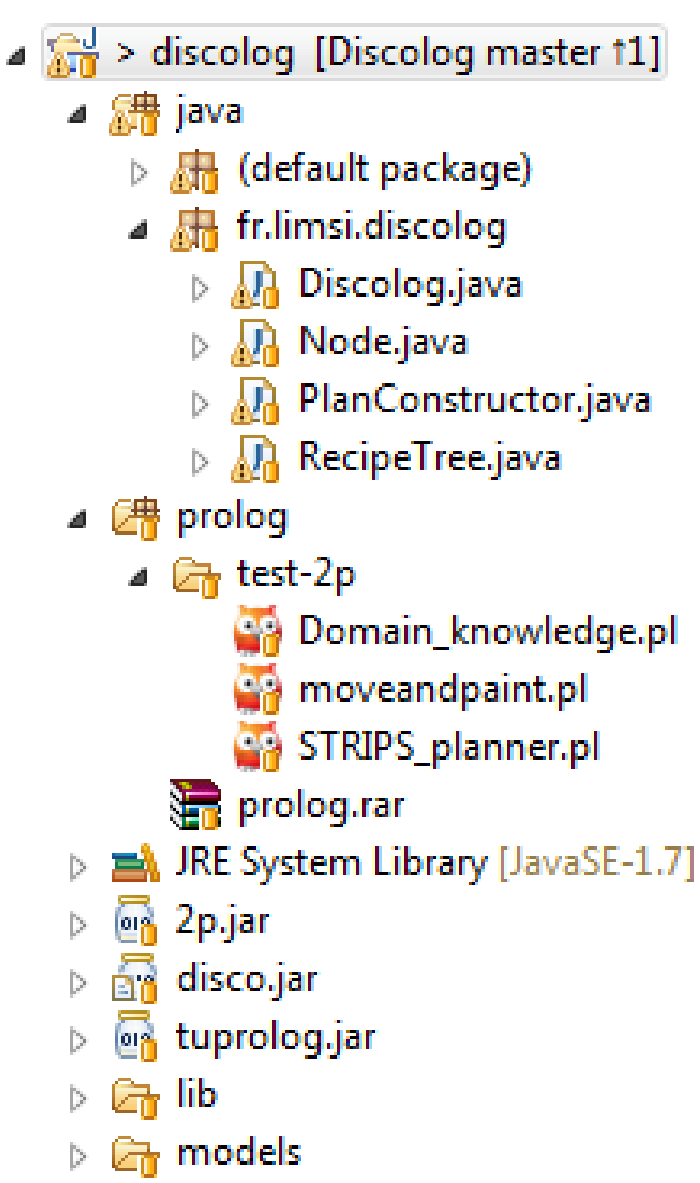

Fig. 5.3: New structure of the Discolog API

5.4 Conclusion

In this chapter, we described the implementation of the Discolog system. First we introduce the environment of the implementation and we introduced

the DISCO system. Next we bring forward the new implemented Discolog API and explain each parts of this later.

in the next chapter, we present the experiments conducted to validate our system.

6. EXPERIMENTS, TEST AND RESULTS

Chapter 6. *Experiments, test and results* The experimental evaluation was devised in two mains parts, the first consisted on constructing the HTN model to evaluate the Discolog system. The second point was to variate the level of knowledge to test the robustness of Discolog system against different types of breakdown.

6.1 experimental model creation

In order to test efficiently the Discolog system, we must test the Discolog system on different type of HTN. in the absence of accurate models. we had to create our own evaluation data. Therefore, an algorithm was constructed to generate different types of HTN models.

The evaluation HTN was constructed using synthetic data as demonstrated in the algorithm ...

- Each compound task in the HTN has a set of $[r_1, ..., r_{\text{recipes}}]$.
- Each recipe is constituted by $[r_1, ..., r_{\text{length}}]$ children to decompose the parent task.
- the preconditions of the first child are the same as its parent, and the postconditions of the last child are the same as its parent.
- Conditions defined in the primitive tasks are chained in each recipe. Example: the task a is decomposed to $\{a_1, a_2, a_3\}$using the recipe R1. Thus, the preconditions of a_1 are the same as its parent a and the postconditions of a_3 are the same as its parent a. We define the postcondition of a_1 as "P1" then the preconditions of the next task a_2 are "P1", we also define the postconditions of a_2 as "P2" then the preconditions of a_3 task are "P2".

- each recipe has its applicability condition, therefore, we generated primitives tasks whose postconditions turns to true the applicability condition of each recipe, in order to be able to recover from a breakdown caused by an applicability condition failure.

Algorithm 2 DiscoLog algorithm

```
procedure CreateHTN(depth,length,recipes,top)
    ConstructHTNTree(depth,length,recipes,top)
    DefineLevelOfKnwoledge(top, level)
EndProcedure

procedure ConstructHTNTree(depth,length,recipes,top)
    if depth > 1 then
        for each r ∈ recipes do
            addRecipe(r, top)
            for each l ∈ length do
                addchild(top,childl)
                ConstructHTNTree(depth, length, recipes, childl)
EndProcedure
```

6.1.1 Test algorithm

The Discolog system was tested on each generated model several times, and for each model, we variate the type of breakdown to recover from and the level of knowledge used in the model. We calculate the percentage of recover relative to the level of knowledge. The generated algorithm is described bellow.

Algorithm 3 Test algorithm

```
loop (10)
    loop(Random breakdowns ← 100)
        (Discolog(HTN,Goal))
    EndLoop

EndLoop
```

6.2 results of the Experiments

APPENDIX

Bibliography