University Name

Doctoral Thesis

Thesis Title

Author:
John Smith

Supervisor:
Dr. James Smith

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

in the

Research Group Name
Department or School Name

September 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Researchers in Artificial Intelligence (AI), 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 planning algorithms for these models. The reasoning about actions and changes is one of the fields in the AI 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 simulated system. However, sometimes during the execution, the current state may not correspond to the expected state. Therefore, the controller can no longer 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 the used devices taking into account the variability of the environment. constructing such model is

time consuming even impossible. Thus, the existing models represent the general view of the world. 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 our work 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.

Chapter 2

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.

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 2.1 .

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

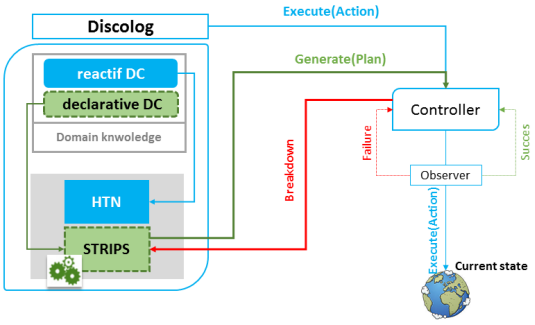

FIGURE 2.1: High level description of Discolog system

planner to propose a plan repair for these candidates. these procedures are described in the next sections.

2.1 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 proceed as follow: As showed in 1, 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 postconditions(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.

2.1.1 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.

2.1.2 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

Algorithm 1 DiscoLog algorithm

procedure DISCOLOG(HTN,Goal)
HTN ← *ConstructModel()*
$\pi \leftarrow$ *Disco(HTN,Goal)*
if $\pi \leftarrow$ *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**
$\Pi += InvokeSTRIPS(candidate, CurrentState)$
$Cost \leftarrow \{cost(\pi) | \pi \in \Pi\}$
return $\pi \in \Pi$ 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

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.

2.2 Example

Lets the HTN describing the move&paint task execution presented in figure 2.2. 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 2.2, 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.

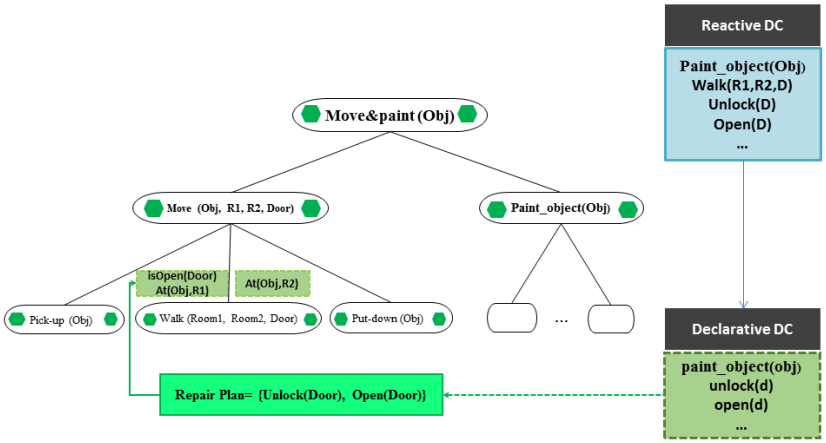

FIGURE 2.2: Plan recovery for the move&paint task using Discolog

Chapter 3

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.

3.1 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 light-weight 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 3.1.

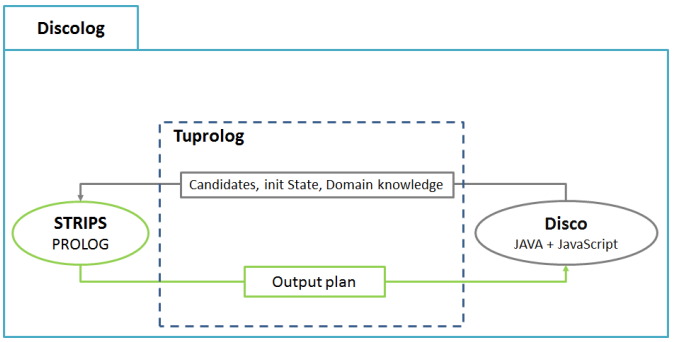

FIGURE 3.1: implementation environment of Discolog

3.2 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 user's 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.

3.2.1 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 3.2 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.

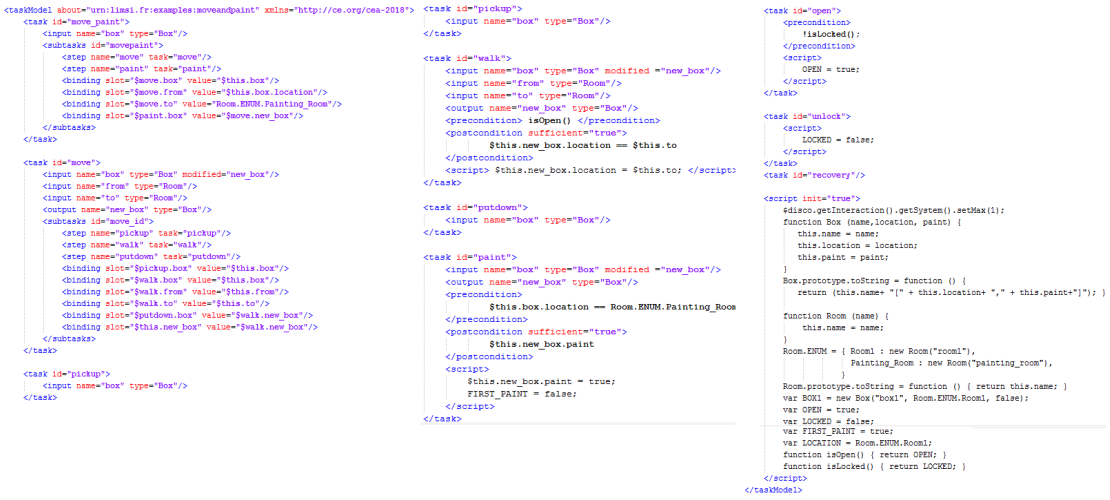

FIGURE 3.2: Complete ANSI/CEA-2018 task model description for the moveandpaint task

3.3 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.

3.3.1 The new Discolog API

In order to implement the hybrid system Discolog we create the API shown in figure 3.3 divided on two mains folders :

3.3.1.1 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;
}
```

3.3.1.2 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.

3.4 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.

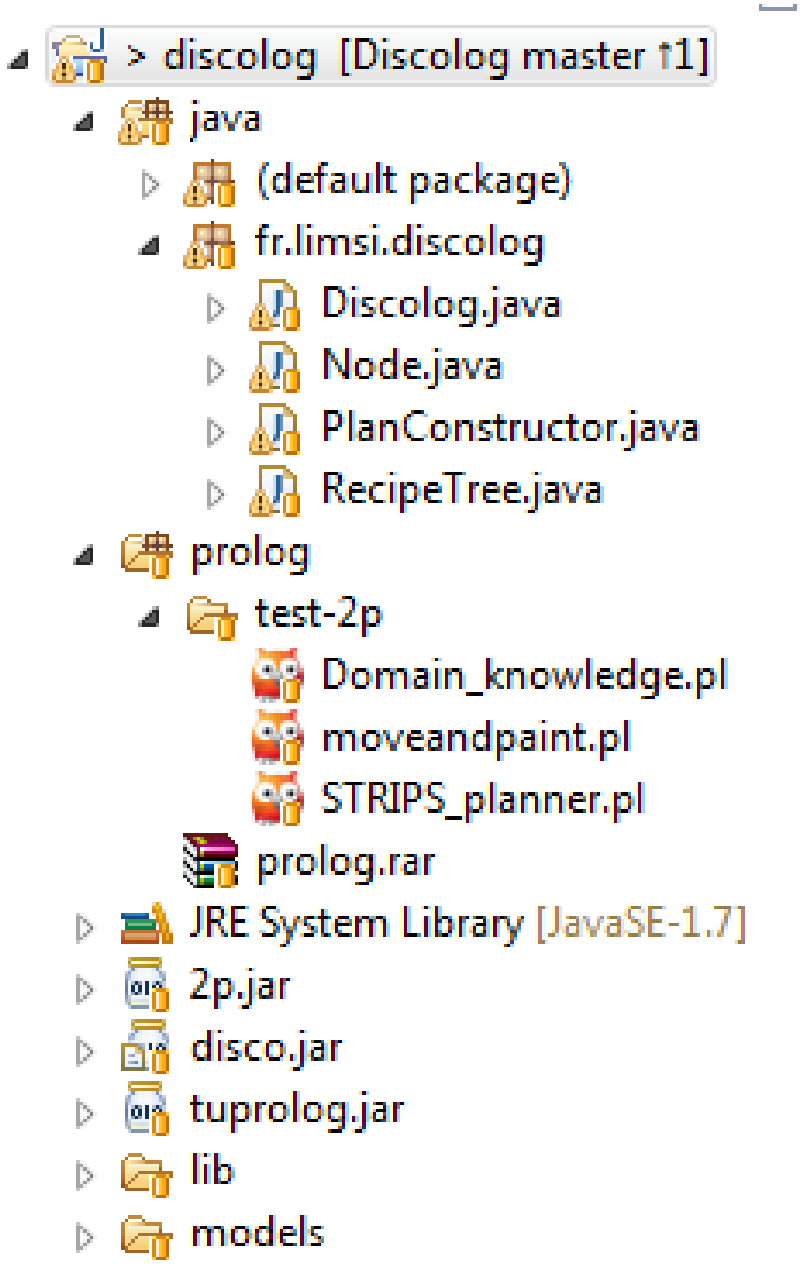

FIGURE 3.3: New structure of the Discolog API

Chapter 4

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.

4.1 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 post-conditions 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
```

4.1.1 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
```

4.2 results of the Experiments