The Thesis Committee for Nitish Sharma
certifies that this is the approved version of the following thesis:

Reactive Synthesis of Action Planners

APPROVED BY

SUPERVISING COMMITTEE:

Mitchell Pryor, Co-Supervisor

Eric van Oort, Co-Supervisor

Reactive Synthesis of Action Planners

by

Nitish Sharma

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

Reactive Synthesis of Action Planners

Nitish Sharma, M.S.E
The University of Texas at Austin, 2018

Supervisors: Mitchell Pryor
Eric van Oort

An increase in the level of autonomy marks one of the fundamental focuses of current robotic systems. This involves the ability of a robot to reason about its environment and plan its motion in order to carry out assigned tasks. For all tasks, it generally involves abstractions into discrete, logical actions, where each discrete action defines a particular capability of the robot.

The problem of synthesis of correct-by-construction action planners has been considered in this work. Action Description Language (ADL) is used to model the actions. These ADL definitions are then translated to Linear Temporal Logic (LTL). LTL based specifications are further used for the reactive synthesis of the strategy.

This work largely focuses on expressiveness which consists of a definition of the actions and system/environment behavior. Classical ADL semantics cannot handle multiple agents or non-determinism. A natural extension of ADL (referred to as ADLnE in this document) has been proposed which can handle dynamic environments, non-determinism, and multiple agents.

The proposed work can be seen as an extension to generic search based action planners. One such A* search-based method, Goal Oriented Action Planner (GOAP) has been considered which is based on ADL semantics and is limited by deterministic, single agent modeling. Through examples, it has been established that for deterministic, single agent and static (or at best quasi-static) systems, the proposed strategy matches that of GOAP. For dynamic and multi-agent situations, a reactive action plan is synthesized (if feasible) that is guaranteed to satisfy the formal specification, i.e. achieve the goal.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Planning can mean different things in different contexts. For action plans, it can be defined as the sequence of steps that must be taken, or the actions/activities that must be performed, for a strategy to succeed. More formally, it can be defined by using state space models [1], where each world instance can be denoted by the state $x \in \mathbb{R}^n$ where n is the number of states and the set of all instances, x, is called the *state space*. Each *action*, a takes the world from one state to another. The model can be defined as follows:

1. A non-empty countable state space set X
2. For each $x \in X$, there exists the action space $A(x)$
3. A state transition function $f(x, a)$ which maps the current x and the action a to new state x'
4. Initial state $X_I \in X$
5. Goal state $X_G \in X$

This is the underlying formulation of the graph search based action planners discussed in Section 2.2.

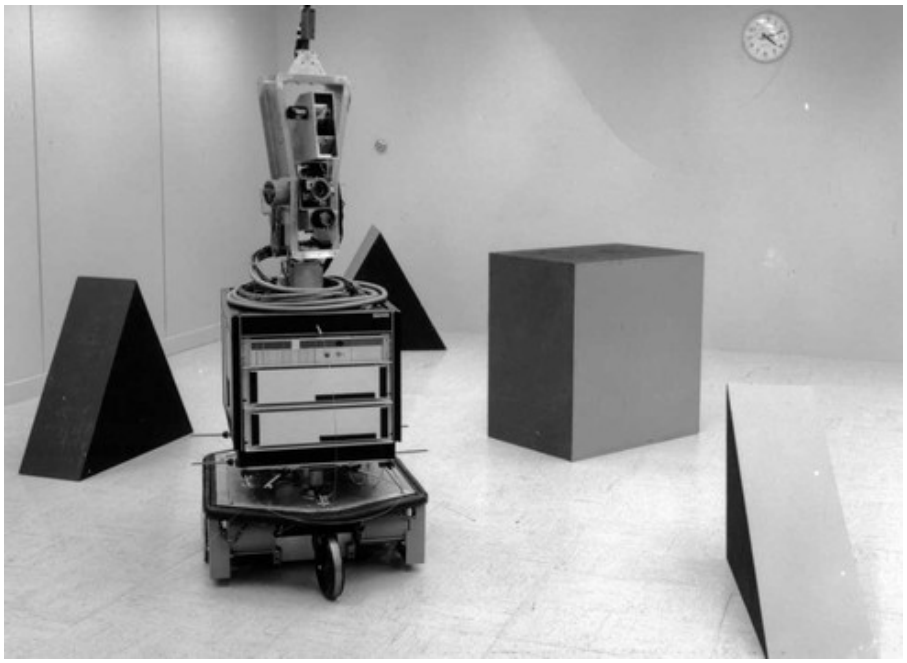


Figure 1.1: Shakey: the first general purpose robot

The state space or state transition models can be specified by using the action languages (commonly known as action formalism) [2]. Action formalism is at the heart of most of the modern-day Artificial Intelligent (AI) systems and dates to the inception of robotics. The early 1970s saw the introduction of Stanford Research Institute Problem Solver (STRIPS) [3]. It was first implemented on Shakey shown in Figure 1.1. STRIPS is considered a stepping stone for the improved action formal languages like Action Description Language (ADL) [4] and Problem Domain Description Language (PDDL) [5]. In these languages, a world is modeled by a set of well-formed formulas (wffs). Operators are the basic elements which form a word in these languages. In our context, each operator corresponds to an action routine whose execution

causes the robot to take certain actions. For example, a robot wants to move from one room to another separated by a door. It will have a routine to look for the door/door-knob, a routine to move near the door, a routine to rotate the knob and so on. Action Planners use this information to calculate the sequence of actions, also known as *plan* or *policy*. A more detailed description is detailed in Chapter 2.

Roughly, the robot control can be divided into two parts: low-level control (action execution) and high-level control (decision making). While the former is essential and in-charge of the individual tasks, high-level control stitches together these individual tasks to perform any complex task. One of the key strengths of action planners lies in the fact that they require only the capabilities of the robot, environment behavior and goal condition to do a task. From this, the planner calculates the sequence of actions that take the system from the current state to the goal state. This work largely focuses on the high-level control and action formalism.

The classical action planners found in the literature have found success in a number of domains/applications and also provide an intuitive way of problem formulation. Although all of these planners try to accomplish goals, none of these provide any guarantees about the plan. These guarantees can include the safety guarantees and the eventuality of the goal condition. To the best of our knowledge, there has not been any work dedicated to providing correct-by-construction action planners. In this work, we take a step towards analyzing this gap.

This work is mainly motivated from a routine inspection task in a non-static, operational, multi-agent environment. Consider a typical rig well platform for drilling [6] as shown in Figure 1.2. An inspection robot can be tasked with inspection, maintenance, or maybe full operational in-charge. Consider a simple task of navigating from one building to another. It's not expected that the tracks will be perfect for robot locomotion. The wheels of the robot could slip on gravel surfaces leading to erroneous navigation commands. Further due to multiple agents (co-operative or adversary) sharing the same work-space, the capabilities provided by classical planners are quite limiting.

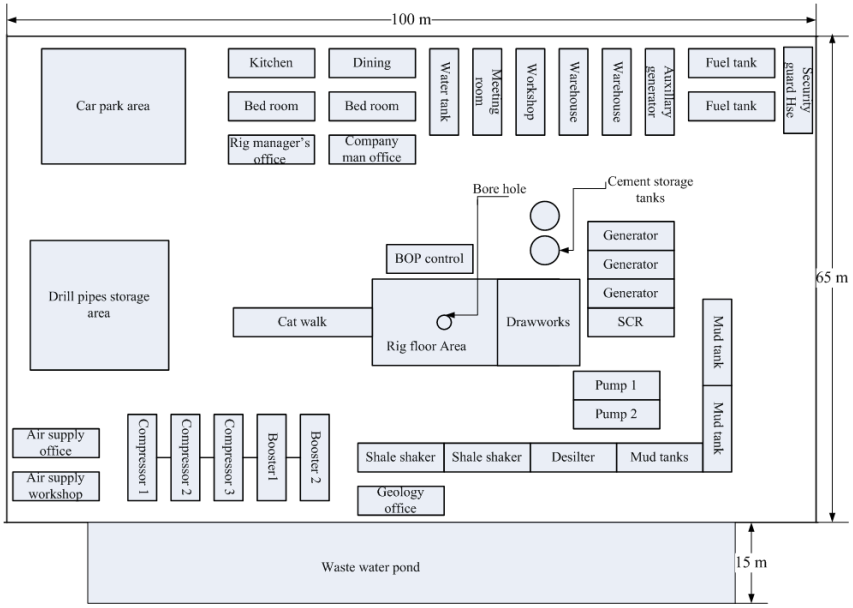


Figure 1.2: A typical rig well platform layout

There are two main contributions of this work. One is the simplicity of the encoding language which is easily understandable by a non-expert operator. We are proposing ADLnE (a natural extension to ADL) which is detailed

in Chapter 3. Another is the translation of ADLnE to LTL, so that we can formulate the problem as reactive synthesis. Reactive synthesis gives us the full proof that the action plan (if feasible) will eventually achieve the goal no matter how the environmental agents behave [7].

The organization of this thesis is as follows. Chapter 2 provides the summary of some of the concepts used in this work. In this chapter, ADL formalism is detailed followed by an A* based planner, GOAP. We also try to point out the limitations in classic planners in Section 2.4. Further, we give an introduction to LTL, reactivity and a simple example for completeness. Chapter 3 tries to fill the gap pointed out in Section 2.4 and highlights the main contributions of this work. It gives the modified semantics for action as ADLnE and translation laws to convert it to LTL. Chapter 4 describes the acStion planning examples where the proposed approach is applied and the results are presented. Through examples, we also establish the expressiveness of the proposed framework and can thus compare it against the current practice. Chapter 5 and 5.2 present the major conclusions drawn from this work and discusses the issues/improvements that remain to be addressed in future.

Chapter 2

Background and Related Work

This chapter presents the necessary background for this work as well as summarizes recent research activities related to action planners. We start by reviewing the abstraction and model predictive definition of action planners. In Section 2.1, we give the formal definition of action in Action Description Language (ADL) schema. Section 2.2 and 2.3 discuss the two of the most popular approaches for solving action planning problems: the search-based planning and SAT-based planning respectively. These have been shown to work well in deterministic planning [8] and have been the center of research in planning community due to their competitive run-time. Section 2.4 talks about the assumptions/limitations in the classical planners which pave the path for this work. This work is built on the LTL (Linear Temporal Logic) based reactive synthesis which is presented in Section 2.5.

The planning problem is often represented using an abstraction to describe the world along with a logical formula to describe the actions. Though it is not explicitly handled in this work, [9, 10, 11, 12, 13, 14, 15, 16] can be referred to for abstraction of the problem domain (low-level tasks, navigation space etc). Bhatia et al. [10], Kress-Gazit et al. [11, 12] handles this as the

partitions of the space according to the areas of interest. Kloetzer and Belta [13], Wongpiromsarn et al. [14], Livingston et al. [15], Wolff et al. [16] decomposes the work-space to abstract the continuous problem to a discrete one. In this work, we assume the input to the planner is already in an abstracted discrete form.

It is convenient to represent the planner as a model predictive control system. Figure 2.1 [17] shows the conceptual definition of dynamic planning. We abstract the world as "World Model" and each instance of this model is called the "state". The Planner outputs the "Plan" which guides the system to move from an initial state to a goal state. The controller guides the system to take a particular action for that instance of the world model. Depending upon the accuracy of the abstraction/model of the world, unexpected events (modeled as interrupts in example 4.1 and [18]) can be handled by re-planning.

2.1 Action Definition

ADL [4, 19] is a commonly used action formalism for representing and reasoning about the effects of an action. It is based on the state-transition model where the effect of an action is characterized as a transition in the world from one state to another. For instance, the effect of the motion of a robot from one location (loc_i) to another location (loc_j) corresponds to the change in the state of robot position from loc_i to loc_j.

More formally, an action "a" over a set of states S is a binary relation a $\subseteq S$ x S such that $\langle s, t\rangle \in a$ iff it is possible for a transition to occur from state

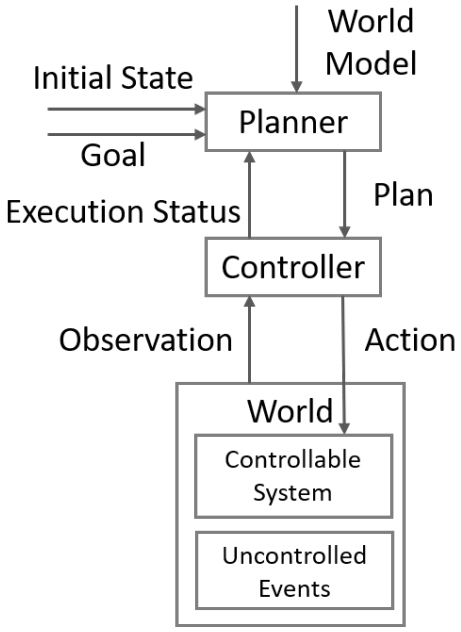


Figure 2.1: Conceptual representation of Planning

s to state t when an action a is performed in state s. Further, a state-transition model is a pair $\langle S, A \rangle$, where S is a set of possible states of the world and A is a set of actions over those states. The state-transition model enforces the (pre-)conditions under which the actions can be performed.

Definition 2.1.1. Schema of Action in ADL: ADL relates the two states (current and next) through one action and can be defined by using 3 terms: predicate + preconditions + effects.

e.g. **Action**(Eat,
PRECOND: Hungry, haveFood
EFFECT: ¬ Hungry)

In the above example, we are defining an action "Eat" which requires that you have the food and you are also hungry. Once the action "Eat" is performed (i.e. you ate the food), you are "not ($\neg$) Hungry" anymore.

One of the characteristics of ADL is that there is no explicit definition of time and all the actions are modeled as discrete events moving the world from one timestamp to another. For an action that is performed at timestamp "t", its pre-conditions should hold at "t" and once the action is done, its effects should be reflected in timestamp "$t + 1$". This property of ADL will be used in Chapter 3 as a translation to the temporal definition of LTL (refer Section 2.5.1). Pre-conditions are the propositions that must hold when an action is performed. Effects are the propositions set by the action.

ADL can be used as an input interface for a number of action planners. Given the initial conditions, ADL/STRIPS action definitions and the goal conditions, we have the complete problem that can be fed to a planner. One such planner is explained in Section 2.2.1.

2.2 Search based Action Planning

A lot of work has been done in this domain mainly because of the advancements in the searching algorithms. All the planners in this category view the world as a graph. Nodes are the instantiation of the world states and the edges capture the actions allowed from each node (state). Then the planner uses graph search techniques such as Breadth First Search (BFS), Depth First Search (DFS), Dijkstras and A^* etc to find a solution path to the

goal states from the current state. One of the main advantages of search-based planners is that additional information could be incorporated via heuristics, most notably the FastForward [20] and FastDownward [21] planners. There are domain specific heuristics [20, 21, 22] that can speed up the search if the information is known about the particular domain.

2.2.1 GOAP

There is no unique method to program a robot. For an industrial fixed task robot, it is enough to control it through a linear script of commands to be performed in a fixed order. But for an environment with uncertainties, a more structured approach is required. The 2015 DARPA Robotics Challenge (DRC) saw a number of teams using the high-level robot control frameworks [23]. These approaches were largely based on the Finite State Machine (FSM) based controllers developed by experts. Though these approaches were developed to get more robust behavior in an unstructured environment, unfortunately, most of them were found to be fragile in practice [24]. Additionally, the size of the FSMs quickly becomes untenable when dealing with increasingly large sets of state variables. It is also time-consuming to add new behavior to the system as the system designer must create new transitions between any new state and the existing states.

Some of these limitations are handled by an A* search based approach, GOAP (Goal Oriented Action Planning) [25]. One application of GOAP is in the AI agents in video games [26, 27]. A ROS package "task_planning" [28] was

developed and is in use by the Nuclear and Applied Robotics Group (NRG) at UT Austin. This is based on classical ADL-like syntax which defines the possible system actions in terms of pre-conditions and post-conditions (effects). This package simplifies coding of the tasks greatly by replacing hard-coded behavior sequences that are generally used in FSMs. Since the actions are not coupled to each other as states are in an FSM, it is much simpler to add new behavior to the system. Therefore the designer can implement new behavior with a minimal re-coding. It should be noted that GOAP is not restricted by the ADL formalism and can handle other languages like PDDL. We are considering the ADL in this work because of its simplicity and its use in [28].

Goal Oriented Action Planning (GOAP) is a technique for AI decision-making that produces a sequence of actions to achieve the desired goal state. The primary objects in GOAP are Goals and Actions. Similar to FSMs, the world can be seen as a collection of Boolean state variables. But instead of modeling each state as the behavior of the world (as in FSMs), by using ADL/STRIPS action definitions, the behavior of the world is associated with each action in GOAP through pre-conditions and effects.

The input to the GOAP planner includes the current world state, a previously selected Goal, and a set of Actions. The Goal's properties are modulated with the current world state to produce the goal world state. It's natural to formulate this as a search over state space graph to find the set of Actions which can produce the goal from the current state.

The fundamental algorithm for GOAP is an A* search over the state-

space. As described in Orkin [26], plan formulation is akin to path-finding. In this formulation, the graph nodes are represented by the world states and the graph edges are actions which transform the world between the two states. Cost functions can be defined for each action. For a given goal, GOAP performs the A* search to move from the current state to the goal state which corresponds to the least cost.

2.3 SAT-based Action Planning

A huge body of research has been developed over the last few decades for planning using logic-based representations [17]. These methods exploit the structure that is particular to the representation. One of the initial works in this direction was by Kautz et al. [29] and was further supported by Rintanen et al. [30], Vidal and Geffner [31] and Nabeshima et al. [32]. Further Kautz et al. [33] proposed some of the basics for encoding plans in the propositional logic.

The SAT-based planners convert the states into Boolean formulas and can use the SAT solvers to find the solution. The Planner defines the Boolean formulas at each timestamp and the action predicates to relate those formulas. Then this state-space is explored by the SAT solvers or any model checker for a plan. The planner begins the planning by checking if the formula corresponding to the initial timestamp achieves the goal. If not, the planner asserts that formula, and the transition between the initial timestamp and the first timestamp is established and checks if the goal is satisfied by the first

timestamp. If these assertions can hold at the same time, then a solution is found. Otherwise, the planner continues to deepen this search, until a solution is found.

2.4 Limitations of Classical Planners

Although the classical action planners have found success in a number of domains and provide an intuitive way of problem formulation, these methods suffer from a number of underlying assumptions:

1. **Determinism:** Classical search based algorithms and ADL/STRIPS semantics based modeling don't allow multiple effects from a single state. It assumes the actions to be perfect.

2. **Static:** There are no external events. Only when an action is being performed, the environment undergoes continuous change until the action has been completed. Thereafter, the environment is assumed to be in a static configuration until the initiation of the next action.

3. **Single Agent:** Classical planners assume only one player(team) operating in the environment. Multiple players can be modeled if they are working together to achieve a goal. If the environment has a smart agent working which can act as an adversary, search based planners can't handle them.

4. **Full Observability:** At each timestamp, it's expected that the information about all the propositions is known before taking the decision

5. **Discrete:** Underlying action definition makes the action planning problem discrete. All the states are abstracted out.

In our motivating example discussed in Chapter 1, most of these assumptions do not hold. The wheels on a mobile system could slip on gravel surfaces, or the goal's location may not have been known with sufficient precision; the oil-rig is operational, trafficked, and therefore, non-static; there are multiple (co-operative and/or passive) agents that could be other robots or human operators; and rig environments are highly occluded and dynamic so the entire action plan may not be known *a priori*.

In this work, we will mainly focus on the first three limitations. In Chapter 3, we will extend the ADL formalism to allow non-deterministic/dynamic modeling of actions and adversary environment. Further, we present the translation laws to conver these definitions to LTL specifications.

2.5 Reactive Synthesis

Reactive synthesis is a method for the automatic construction of reactive protocols from logical specifications. We transform a temporal specification (LTL in our context) into an implementation that is guaranteed to satisfy the specification for all the possible behaviors of the environment. Although there are a number of algorithms to achieve this [34], we will only focus on the game-based synthesis (refer to Section 2.5.2) as its performance is asymptotically optimal in the size of the specification.

This section gives an introduction to propositional LTL (Linear Temporal Logic) followed by the formulation of the two-player game. A simple example is also presented for completeness.

2.5.1 Linear Temporal Logic

LTL [35] is a formalism that extends the propositional logic to include the temporal operators. This provides simple and intuitive yet mathematically precise definitions of LT (Linear Temporal) properties which can be inferred as the traces that a system should exhibit. The sense of time captured by the term temporal is abstract and should not be confused with the absolute time. It captures the relative order of the events and is ideal for Action Planner definition.

Atomic Proposition (AP) is the building block of LTL-based formulae and captures the assertions about the variables (e.g. the robot is at loc_1). It can either be true or false. For example, "What is the current location of the robot?" is not an AP. Apart from propositional connectives like conjunction ($\wedge$), disjunction($\vee$), and negation ($\neg$), LTL defines two temporal operators next ($\circ$) and until ($\cup$)

Definition 2.5.1. Syntax of LTL

LTL formulae (used as ϕ and ψ in this work) over the the set of Atomic Propositions (a $\in$ AP) follows the following grammar:

$$\phi := true \mid a \mid \neg\phi \mid \phi 1 \wedge \phi 2 \mid \circ \phi \mid \phi 1 \cup \phi 2 \tag{2.1}$$

which denotes that the LTL formula ϕ can be take the value as true, some literal a, or it can be linear function of other formulae like $\phi 1 \wedge \phi 2$ and $\neg\phi$. Although disjunction($\vee$) is not explicitly defined in LTL syntax, it can be easily derived using conjunction ($\wedge$) and negation ($\neg$). Apart from all the propositional operators, LTL also defines following temporal operators which can be derived by using next ($\circ$) and until ($\cup$):

1. $\Diamond$**:** Will eventually happen in the future

$$\Diamond\phi \equiv true \cup \phi \tag{2.2}$$

2. $\Box$**:** Now and Forever in the future

$$\Box\phi \equiv \neg\Diamond\neg\phi \tag{2.3}$$

3. $\Box\Diamond$**:** Infinitely often in the future

$$\Box\Diamond\phi \equiv \forall i \exists j : \phi, j \geq i \tag{2.4}$$

4. $\Diamond\Box$**:** Eventually Forever in the future

$$\Diamond\Box\phi \equiv \exists i \forall j : \phi, j \geq i \tag{2.5}$$

Definition 2.5.2. Semantics of LTL

The semantics of LTL are defined over words. A word is an infinite sequence of letters from the set 2^{AP}. Each letter (l_i labels the state of the formula at instance 'i'. A word w satisfies a LTL formula ϕ (denoted by $w \models \phi$) if one of the following is true:

1. $\phi \models \text{true}$

2. ϕ is an atomic proposition, and $\phi \in l_0$

3. $\phi = \phi_1 \wedge \phi_2$, $w \models \phi_1$, and $w \models \phi_2$.

4. $\phi = \circ\psi$, $w_1 \models \psi$

5. $\phi = \neg\psi$, $w \not\models \psi$

6. $\phi = \phi_1 \cup \phi_2$, $\exists$ j $\forall$ $0 \leq i < j$, $w_i \models \phi_1$ and $w_j \models \phi_2$

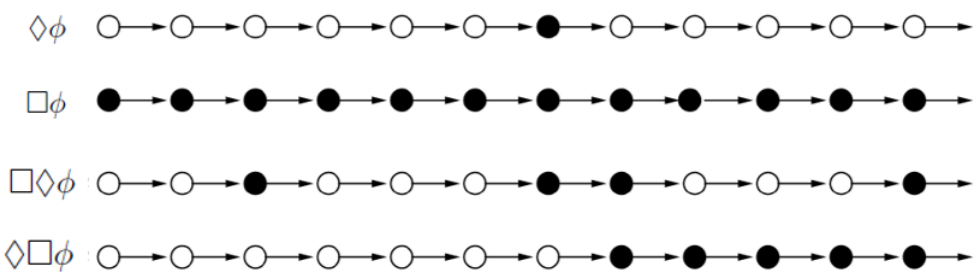


Figure 2.2: Semantics of LTL operators

Figure 2.2 shows semantics of some of the derived LTL operators. The dark circle should be read as the property ϕ being satisfied at that instance. For example, for $\Diamond\Box\phi$, property ϕ holds forever after a certain instance of time. For more formal analysis of LTL, readers are referred to "Principles of Model Checking" by Baier and Katoen [36]

2.5.2 Two Player Game

We analyze our action planner in terms of a two player game. Consider a reactive system in Figure 2.3 where our system (player 2) decides the control

variables (represented as y) and environment decides (player 1) all the other world variables (represented as x). If all the transitions are defined in LTL, the dynamics of the game can be captured as a linear temporal formula $\Phi(x, y)$. Environment attempts to falsify the specification $\Phi(x, y)$ and the plant tries to satisfy it. By considering the environment to be adversarial, the system has a winning strategy iff $\forall x\ \exists y$ such that $\Phi(x, y)$ is realizable.

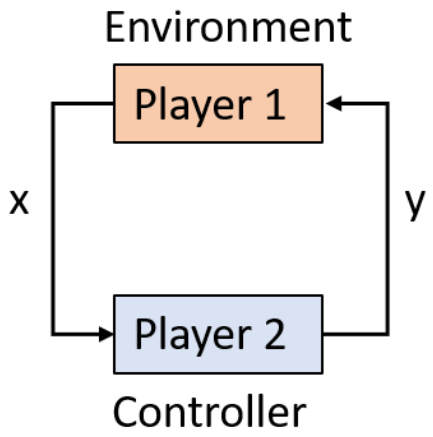


Figure 2.3: Reactive system as a two player game

Definition 2.5.3. Game structure is a tuple [7] $G = (V, X, Y, \theta_e, \theta_s, \rho_e, \rho_s, \phi)$ where,

1. V is a finite set of state variables. Σ_V is the set of all possible assignments to variables in V.
2. $X \subseteq V$ is a set of input/environment variables
3. $Y = V \setminus X$ is a set of output/controlled variables

4. θ_e is atomic proposition over X denoting initial state of input variables

5. θ_s is atomic proposition over Y denoting initial state of output variables

6. $\rho_e(V, \circ X)$ is a transition relation that relates a state and possible next input values

7. $\rho_s(V, \circ X, \circ Y)$ is a transition relation that relates a state, an input value to possible next output values

8. ϕ is an LTL formula characterizing the wining condition

The above definition gives a general LTL game structure. Solutions to generic two-player games are known to have prohibitively high complexity and worst case complexity is 2-EXPTIME (i.e. double exponential). In this work, we are considering a fragment of LTL known as the General Reactivity (GR(1)) [7, 37]. A game structure with GR(1) winning condition can be solved in $\bigcirc(\Sigma_V^3)$ and looks as follows:

$$\begin{aligned} \phi &= ((\phi_q \wedge \phi_e) \rightarrow \phi_s), \\ \phi_\alpha &= \phi^\alpha_{initial} \wedge \Box\phi^\alpha_1 \wedge \Box\Diamond\phi^\alpha_2 \end{aligned} \tag{2.6}$$

where ϕ_q characterizes all the non deterministic transitions of the system, ϕ_e captures all the environment behaviors and ϕ_s is the expected behavior of the controlled system. More precisely, all GR(1) formulae used in this work are of the form given by ϕ_α where $\alpha \in (q, e, s)$. $\phi^\alpha_{intiial}$ is the propositional formula characterizing the initial conditions. ϕ^α_1 is the set of the transition relations characterizing safe, allowable moves, and propositional formulae characterizing

invariants; all the states that are invariant and ϕ_2^α is the set of the propositional formulae characterizing all the states that should be reached infinitely often.

Any state of the game can be represented as $((e, q), s)$. The transition of the game is the move of uncontrolled variables followed by the controller (action planner) move. Planner chooses an action such that the specifications are satisfied. For a winning strategy to exist ($\phi(x, y)$ is realizable), the specification should be met for all the behaviors of the environment. At each state, the system chooses an action, which drives the system to a (number of) possible state(s) due to (non-)determinism. If the specification is realizable, solving the two-player game gives a finite automaton that effectively gives a state-feedback action planner for fully observable non-deterministic system [38]. By observing the state which the system enters, the next action is chosen accordingly by reading the finite automaton.

Given a two-player game and GR(1) specifications, by using off-the-shelf GR(1) synthesis tools [39, 40, 41], one can obtain a finite automaton that represents an action plan for the system. In this work, we are using Slugs [40] for automaton extraction.

Apart from GR(1), there are other LTL fragments that can handle action planners like synthesis from co-safe LTL specifications [42, 43, 44]. Our main reason for choosing GR(1) synthesis is its ability to specify reactivity with respect to a dynamic, and even adversarial (worst-case), environment, such as external events sensed by the robot and low-level system failures

2.5.3 Example

The example presented here is a variation of runner-blocker example [45]. Consider a robot navigating in a T-shaped grid world shown in Figure 2.4. The robot can move one step in any direction per time step. There is a dynamic obstacle that our robot needs to dodge. Obstacle is modeled as non-deterministically navigating in middle column (s_1, s_3, s_4). Our robot starts from s_0 and it needs to get to s_2. For this example, let's assume the robot never collides with the walls. A similar example will be revisited in Section 4.3 where we also model the collision and the controller will assure no-collision as a safety property. Figure 2.4 shows the initial state of the system. Robot starts from $(y = s_0)$, and the obstacle is at the position $(x = s_3)$.

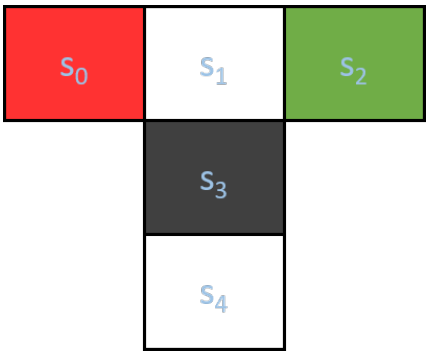


Figure 2.4: T-shaped grid world

Two-player game tuple for this system can be written as:

1. $X := \{x\}$, $\Sigma_X = \{s_1, s_3, s_4\}$

2. $Y := \{y\}$, $\Sigma_Y = \{s_0, s_1, s_2, s_3, s_4\}$

3. $\theta_e := (x = s_3)$

4. $\theta_s := (y = s_0)$

5. $\rho_e := \Box\ (\ (x = s_3 \implies x \neq s_3) \wedge (x \neq s_3 \implies x = s_3)) \wedge \Box\Diamond(x = s_4)$

6. $\rho_s := \Box\ (\ (y = s_0 \implies (y = s_0 \vee y = s_1))$
   $\wedge\ (y = s_1 \implies (y = s_1 \vee y = s_0 \vee y = s_2 \vee y = s_3))$
   $\wedge\ (y = s_2 \implies (y = s_2 \vee y = s_1))$
   $\wedge\ (y = s_3 \implies (y = s_3 \vee y = s_1 \vee y = s_4))$
   $\wedge\ (y = s_4 \implies (y = s_4 \vee y = s_3))$
   $\wedge\ (\circ x \neq \circ y) \wedge (x \neq \circ y))$

7. $\phi := \Diamond(y = s_2)$

Figure 2.5 shows the state transition diagram of our system. The red line transitions capture the move of the agent and black line transitions show the environment behavior. Black nodes are the runs where the robot loses against the environment. Green nodes are the winning runs. Each trace should be read as the word over (x, y) i.e. $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)....., (x_i, y_i), ...\}$

Figure 2.6 shows the winning strategy for the controller against all the possible environment behaviors. It should be noted that though there are two winning nodes in state transition diagram but reactive synthesis prunes out the right winning node as it models the environment as adversary. Sample trace for right winning node is $\{(s_3, s_0), (s_4, s_0), (s_4, s_1), (s_3, s_1), (s_3, s_1), (s_4, s_1), (s_4, s_2)\}$ which is possible only if the environment assists our agent in achieving the goal

and doesn't act as an adversary. If the environment acts smartly, it will follow the $\{(s_3, s_0), (s_4, s_0), (s_4, s_1), (s_3, s_1), (s_3, s_1), (s_1, s_1)\}$ to fail the strategy for the same controller moves.

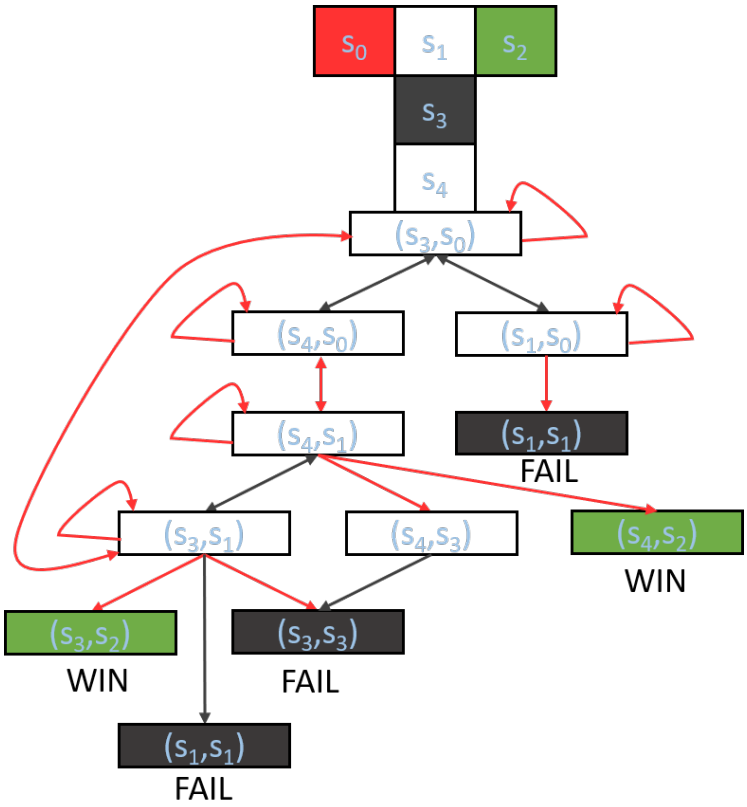


Figure 2.5: State Transition Diagram of two player game

It is possible to extract a strategy iff $\forall x \exists y \Phi(x, y)$ is realizable. In our example, if we don't have the fairness assumption $(\square\Diamond(x = s_4))$ on the environment, the environment can keep the controller stuck in the word $\{((s_3, s_0), (s_1, s_0), (s_1, s_0), (s_3, s_0))^\omega\}$ forever failing the goal. It should be noted that if the environmental agent is acting as adversary, only if it's at $(x = s_4)$

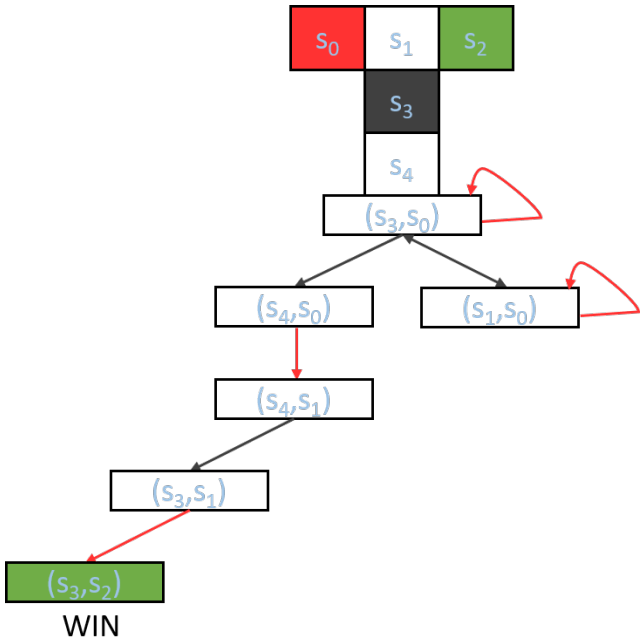


Figure 2.6: Winning strategy for the controller

block, our agent can move to $(y = s_2)$ and then to the goal condition. In all other scenarios, the external agent can collide with our agent. By introducing the fairness assumption $(\square\lozenge(x = s_4))$ on the environment, we are assuming that sometime in future the environmental agent will visit the $(x = s_4)$ block which makes the strategy realizable.

In this Chapter, we started with the formal definition of action in Action Description Language (ADL) schema. We then gave an overview of state of the art action planners: search-based planners and SAT-based planners. All these methods are known to work well in literature but it is also well established that these are plagued with a number of assumptions which are highlighted in

Section 2.4. In this work, we are mainly focusing on non-determinism, dynamic and multi-agent modeling. To alleviate these limitations, we will be using the reactive synthesis of *correct-by-construction* action planners. GR(1) fragment of LTL provides an intuitive and expressive to formulate the problem as two player game. In Chapter 3, we will provide the translation laws to convert a action planning problem defined in terms of action definitions into reactive synthesis formulation.

Chapter 3

ADLnE and its GR(1) translation

In Section 2.4, some of the limitations of the classical action planners were highlighted. Most of those are rooted to the semantics of ADL/STRIPS which doesn't allow non-determinism, can only handle co-operative agents in the multi-agent problems, and the environment is modeled to be discrete, static, and fully observable.

In this work, we target the non-determinism, multi-agent modeling in a dynamic environment by extending the ADL to ADLnE (Action Description Language natural extension). The following section gives the modified definition of action as handled in ADLnE and proposes a few properties which relate the expressibility of ADL with the ADLnE. Further in Section 3.2, translation laws are presented which can be automated to translate the planning problem formulated in ADLnE to the reactive synthesis defined in GR(1) fragment of the LTL.

In this work, we are constructing the formal specifications from the action definitions supplied by user input in ADLnE. We could have asked the user to directly write all the specifications in LTL (which will be more expressible) but we are focusing on the simplicity of the input. There has

been similar work handling natural language [46], structured english [47] and multi-paradigm specifications [48]. Although the imperative element of these works could have been really helpful, they are not necessary for the current focus and can be explored for future integration.

3.1 ADLnE action definition

ADLnE is also based on the same state-transition model as ADL. However, we extend it to handle the controller and the environmental agent's actions separately. Each action of either the system or the environment(adversary) is characterized as a transition in the world state. ADLnE adds two more features to the ADL action schema (refer to definition 2.1.1):

1. The `agent_tag` which captures the nature of the action and will be used in translation laws (Section 3.2) to differentiate between the controller and the environment. It can have two possible values:

   (a) **SYS**: This tag can be used to define the system/controller capabilities. For all the controller actions that can be performed in parallel, use the separate tags like SYS_1, SYS_2 and so on.

   (b) **ENV**: This tag can be used to model the environment/adversary behavior. Similar to the SYS tag, the parallel actions should be declared with separate tags.

2. Non-deterministic modeling of:

(a) **Pre-conditions**: It is possible to perform the same action with the different pre-conditions. e.g. the same action can be performed from different locations. Although this functionality can be reproduced in the ADL by using different action names, ADLnE helps in the compact formulation of the same problem.

(b) **Effects**: Actions are modeled as deterministic operators in the classical planners i.e. they are considered to be perfect. This doesn't capture the realistic behavior. ADLnE instead considers the actions to have multiple effects which can be modeled as adversaries for the system and used to analyze the worst case scenario.

Definition 3.1.1. Schema of Action in ADLnE: ADLnE relates the two states (current and next) through the `agent_tag` and its action. It can be defined by using 4 terms: action_name + agent_tag + preconditions + effects. Action definition in ADLnE looks as follows:

Action(action_name, `agent_tag`,
PRECOND: $\{cond_1\}$, $\{cond_2\}$, ..., $\{cond_n\}$
EFFECT: $\{effect_1\}$, $\{effect_2\}$, ..., $\{effect_m\}$)

Pre-conditions and effects can be modeled as non-deterministic propositional formulas in this schema. Although any propositional formula is possible to define the pre-conditions and effects and it's possible to convert all logical formulas into an equivalent disjunctive normal form [49], we are assuming them to be in Disjunctive Natural Form (DNF) to be explicit. DNF is a logical

formula which is a disjunction of conjunctive clauses. Each clause $\{\text{cond}_1\}$ or $\{\text{effect}_1\}$ does not have any disjunction among its atomic propositions.

In the example below, the intended action is to move from location (x=4,y=4) to (x=4,y=6) but as the action is not perfect, the agent can end up either at (x=3,y=6) or (x=4,y=6) or (x=5,y=6). This kind of modeling is not permissible in ADL semantics.

e.g. **Action**(moveNorth2Steps, SYS_1,
PRECOND: {at_x_4_y_4}
EFFECT: {at_x_4_y_6}, {at_x_5_y_6}, {at_x_3_y_6})

The following properties establish the relationship between ADL and ADLnE. Proposition 3.1.1 comments on the expressiveness of ADLnE. Though we are not providing a formal proof for these properties, following analysis provides an intuition into their validity. Consider a set of models, M. For a formula $\psi \in ADL$, such that $\psi \models m$ $(m \in M)$, it can be seen that there always exists a formula $\phi \in ADLnE$ such that $\phi \models m$. ADLnE can be considered as a superset of ADL and the relation among their formulas can be represented in equation 3.1. For more formal analysis of expressiveness, please refer [50].

$$\begin{aligned} \psi \models m \Rightarrow \phi \models m \\ \phi \models m \nRightarrow \psi \models m \end{aligned} \tag{3.1}$$

Proposition 3.1.1. ADLnE is more expressive compared to ADL

Corollary 3.1.2. An action planning problem defined in ADLnE with all the actions satisfying following properties is equivalent to ADL based formulation.

1. All the actions have agent_tag = SYS
2. Preconditions don't have any disjunction joining state literals
3. Effects don't have any disjunction joining the state literals

Corollary 3.1.2 derives the ADL formalism from ADLnE semantics. It describes the scenario of a single agent (or collaborative multiple agents working in sequential order) operating in a deterministic and static world. We will revisit this Corollary in example 4.1 in which we will reproduce the GOAP action planning problem in the ADLnE framework and establish the equivalence. By removing the second condition in Corollary 3.1.2, we can model the single agent with imperfect actions operating in a static world.

3.2 ADLnE to LTL conversion

Given the initial conditions, ADLnE action definitions and the goal conditions, we have the complete problem definition that can be used by the planner. We want to formulate our problem as the reactive synthesis to automate the task of correct-by-construction FSMs. Though most of the synthesis solutions are highly complex to handle (2EXPTIME), General Reactivity (GR(1)) [7, 37] can be solved in $\bigcirc(\Sigma_V^3)$ (where Σ_V is the set of all possible assignments to set of state variables V).In this work, we are using Slugs [40] for automaton extraction as the authors are most familiar with this tool at this time. However, other GR(1) synthesis tools like Tulip [39], and Ratsy [41] etc are also available and can be explored in the future.

Section 2.5.2 explains the general structure of the two player game and the GR(1). Slugs provides the input file format which uses the headers like "[INPUT]", "[OUTPUT]", "[SYS_INT]", "[ENV_INT]" which have the one-to-one mapping with the elements of the tuple in the definition 2.5.3. In the following discussion, we expand on the transition relations ρ_s and ρ_e to relate them to the Slugs headers.

1. ρ_s is the transition relation that captures the relation between current state and all the possible next states controlled by the system. GR(1) allows two temporal operators:
   (a) Always ($\square$): All the system safety guarantees are included under this category which means the conjunction of all the LTL formulas that the system should satisfy at all timestamps. In our context, this includes the safety properties (e.g. noWallCollision property defined in Section 4.3.1), and the pre-conditions. In Slugs, it is used under the header name "[SYS_TRANS]".
   (b) Always Eventually ($\square\lozenge$): All the system liveness guarantees are included under this category which means the LTL formulas that the system should satisfy infinitely often or once in a while. We will use this to encode the goal conditions. If our game is of finite run (goal is the terminating condition), it can be easily seen that $\square\lozenge \iff \lozenge$ (refer Section 4.1). In Slugs, it is used under the header name "[SYS_LIVENESS]".

2. ρ_e is an equivalent of ρ_s from the environment perspective. Its expressiveness is also limited to two temporal operators by GR(1):

   (a) Always ($\Box$): All the environment safety assumptions are included under this category i.e. the constraints on the behavior of the environment. All the inputs that are not constrained, the environment has full autonomy to drive them to any legal value (defined by the domain of the variable). In Slugs, it is used under the header name "[ENV_TRANS]". We will use this header to model the environment behavior and the action effects.

   (b) Always Eventually ($\Box\Diamond$): All the environment liveness assumptions are included in this category. This operator is used to model the fairness properties of the environment. In Slugs, it is used under the header name "[ENV_LIVENESS]".

A planning problem has two main components: world state variables and action. In our formulation, we assume all the world state variables are controlled by the environment and are used as inputs to our planner. In Slugs, they are encoded under the header "[INPUT]". On the other hand, Actions are modelled to be controlled by the system and are used as inputs to our planner. In Slugs, they are encoded under the header "[OUTPUT]"

From the schema of ADLnE 3.1.1, we know that an action can be defined by 4 terms. Using this definition of action, we lay down the following

laws that can be used to translate the ADLnE to LTL and hence, formulate the planning problem as reactive synthesis. For a given `agent_tag`,

1. Enumerate all the actions under each `agent_tag`. For example, if there are 5 actions that our system can take. Then use action: 0...5. action = 0 is used to model the scenarios when we don't need any action like after we have achieved the finite run goal. Refer to Chapter 4 for better representation of this rule. If the actions are controlled by the system (`agent_tag` = SYS), they are modeled as the output of our planner. In Slugs, they are encoded under the header [OUTPUT]. If the actions are controlled by the environment (`agent_tag` = ENV), they are encoded under the header [INPUT] with the world state variables.

2. If a variable is in the effect list of an `agent_tag` (say SYS_1) but not in the effect list of an action (also defined for SYS_1), it should remain unchanged (refer Section 4.1.2 for example).

3. If an `agent_tag` has only one action, and pre-condition set to TRUE, action_name can be dropped while modeling the action effects. These conditions can be considered as safety guarantees if given under SYS `agent_tag` and as safety assumptions if given under ENV `agent_tag`.

4. Pre-conditions should hold in the same time stamp when the action is started i.e. $\forall a, a \Rightarrow PRE(a)$ where PRE(a) is the set of pre-conditions for action a. If an action has `agent_tag` = SYS, it should be encoded

in [SYS_TRANS], otherwise, it should be encoded under [ENV_TRANS] header. As Slugs uses the end of transition (denoted by adding a ' to the variable name) as the constraint for synthesis, we will use the $a' \Rightarrow PRE(a)'$ to encode this property.

5. Effects should hold at the end of the transition i.e. $\forall a, a \Rightarrow EFF(a)'$ where EFF(a) is the set of effects for action a. These are encoded under the [ENV_TRANS] header no matter what the `agent_tag` is.

In Chapter 4, we will use the above rules to translate the planning problem with the actions defined in ADLnE to reactive synthesis. Sample runs will also be presented to show the soundness of this approach.

Chapter 4

Use cases

4.1 Deterministic system

This section establishes the Corollary 3.1.2. We use the GOAP (Goal Oriented Action Planner) framework here for comparison with ADLnE. The example is motivated from the inventory inspection operation presented in [18] and Figure 4.1 shows the original robot used in the survey. The Pioneer LX features greater payload and a more extensive sensor suite, including an Asus depth sensor, RGB camera on a pan tilt unit and ultrasonic forward and rear sensors.

Figure 4.1: The Adept Pioneer LX mobile robot platform

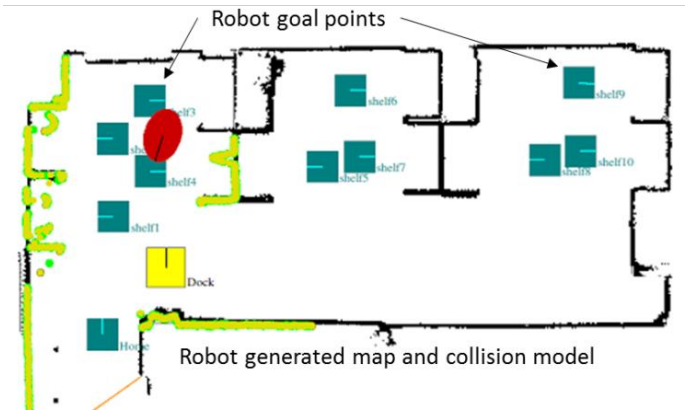


Figure 4.2: Diagram of the vault with a number of cabinets used in the original experiment

Figure 4.2 shows the layout of the mock-up vault and Figure 4.3 shows the simplifications made to the vault for this example. There are n shelves that the robot needs to go and survey. It starts from its docking area and needs to return back to the docking area after surveying all the shelves. For this example, we are assuming there are only 5 shelves. The world is defined by using the following atomic variables:

1. docked: Is the robot in the docking station?
2. at_dockingArea: Is the robot near the docking area?
3. battery_low: Is the battery charge below the low threshold?
4. at_cabinetX: Is the robot in front of the cabinet X (X $\in \{1, 2, 3, 4, 5\}$?
5. cabinetX_surveyed: Has the cabinet X been surveyed?

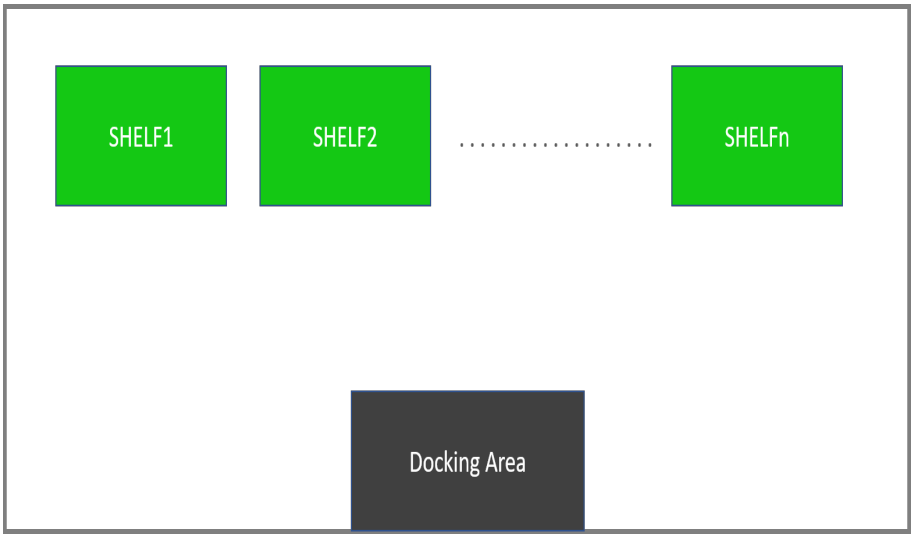


Figure 4.3: Mock-up Vault under survey

4.1.1 GOAP formulation

For GOAP, we are using NRG's "task_planning" [28] ROS package. The following classes implement the GOAP algorithm:

1. Worldstate - Stores a vector of Boolean variables (atoms) which represent the state of the world and robot.

2. Action - Modifies the world atoms in some way. An action is defined by its preconditions, postconditions, and cost function.

3. ActionPlanner - Stores the set of actions and produces action plans using the createPlan() function.

4. TaskManager - Top level class providing both the task planning and the execution.

As pointed out in action definition 2.1.1, we only need pre-conditions and effects to define an action in ADL. All the actions in ADL can be defined as follows:

Action(dock,
PRECOND: { ¬ docked, at_dockingArea }
EFFECT: { docked})

Action(unDock,
PRECOND: { docked, ¬ battery_low }
EFFECT: { ¬ docked})

Action(moveToX,
PRECOND: { ¬ docked, ¬ at_cabinetX, ¬ battery_low }
EFFECT: { at_cabinetX, ¬ at_dockingArea, ¬ at_cabinetY })
(Y $\in \{1, 2, 3, 4, 5\} \setminus$ X)

Action(moveToDockingArea,
PRECOND: { ¬ at_dockingArea }
EFFECT: { at_dockingArea, ¬ at_cabinetX }) s.t. (X $\in \{1, 2, 3, 4, 5\}$)

Action(surveyX,
PRECOND: { at_cabinetX, ¬ cabinetX_surveyed, ¬ battery_low }
EFFECT: { cabinetX_surveyed }) s.t. (X $\in \{1, 2, 3, 4, 5\}$)

Action(chargeBattery,
PRECOND: { docked, battery_low }
EFFECT: { ¬ battery_low })

Using the initial and final states from Table 4.1, the "action stack" returned by GOAP is:

1. unDock
2. moveTo1
3. survey1
4. moveTo2

   .....

   .....

11. survey5
12. moveToDockingArea
13. dock

Table 4.1: Initial and Goal Conditions for vault survey

Atom	Initial	Final
docked	True	True
at_dockingArea	True	don't care
battery_low	False	don't care
at_cabinetX	False	don't care
cabinetX_surveyed	False	True

The plan produced by GOAP is quite intuitive which we would expect from any rational system (or human supervisor). The robot should start by un-docking itself, move to all the shelves one by one followed by the survey of each shelf, move back to the docking area, and dock itself again satisfying

the goal conditions. Furthermore, it should recognize when certain shelves are not accessible or the battery is low and then take appropriate actions before attempting to progress towards the goal condition. Note that two external control loops are also included in the Pioneer LX software, but are not integrated with GOAP. These monitor for collisions and high radiation levels. A collision simply pauses the robot causing it to wait until the obstacle clears or it recalculates a valid motion plan. If high levels of radiation are detected, then an audible alarm is sounded and the robot will not progress in the plan produced by GOAP until the alarm is manually cleared.

4.1.2 ADLnE formulation

In this example, we are assuming that the robot is the only agent operating in the static/deterministic environment with its set of perfect actions. From Corollary 3.1.2, we expect both representations (ADL and ADLnE) to be equivalent, if ADLnE uses agent_tag = SYS. Some of the examples of the ADLnE action definitions are as follows:

Action(dock, SYS
PRECOND: { ¬ docked, at_dockingArea }
EFFECT: { docked})

Action(surveyX, SYS
PRECOND: { at_cabinetX, ¬ cabinetX_surveyed, ¬ battery_low }
EFFECT: { cabinetX_surveyed }) s.t. (X $\in \{1, 2, 3, 4, 5\}$)

Action(chargeBattery, SYS

PRECOND: { docked, battery_low }

EFFECT: { ¬ battery_low })

Other actions can be easily represented in ADLnE in a similar way. There are 14 actions (6 moveTo + 5 survey + dock + undock + charging) that the controller can take. By using the translation laws defined in Section 3.2, we can convert these definitions to LTL and formulate it as reactive synthesis. Further, we can extract the automaton which captures the states of the world. Following is the Slugs [40] encoding for this example:

```
[INPUT]
# Atomic variables
docked:0...1
battery_low:0...1
## robot location
at_dockingArea:0...1
at_cabinet1:0...1
at_cabinet2:0...1
at_cabinet3:0...1
at_cabinet4:0...1
at_cabinet5:0...1
## survey status of the cabinets
cabinet1_surveyed:0...1
cabinet2_surveyed:0...1
cabinet3_surveyed:0...1
cabinet4_surveyed:0...1
cabinet5_surveyed:0...1

[OUTPUT]
## 0 = No acton,
# 1  = moveTo1, 2 = moveTo2, 3 = moveTo3, 4 = moveTo4, 5 = moveTo5,
# 6  = moveToDockingArea
# 7  = survey_cabinet1, 8 = survey_cabinet2, 9 = survey_cabinet3,
# 10 = survey_cabinet4, 11 = survey_cabinet5,
# 12 = unDock, 13 = dock, 14 = charge_battery
action:0...14

[SYS_INIT]
## No initialization of action
```

```
[ENV_INIT]
## Initialization of environment variables
docked = 1
cabinet1_surveyed = 0
.....................
at_cabinet1 = 0
.....................
at_dockingArea = 1
battery_low = 0

[SYS_TRANS]
## Pre-conditions for the actions
(action' = 1  -> ( battery_low' = 0 & docked' = 0 &
                   at_cabinet1' = 0))
.............................
(action' = 6  -> ( at_dockingArea' = 0))
(action' = 7  -> ( battery_low' = 0 & at_cabinet1' = 1
                  & cabinet1_surveyed' = 0))
.............................
(action' = 12  -> ( battery_low' = 0 & docked' = 1 ))
(action' = 13  -> ( docked' = 0 & at_dockingArea' = 1))
(action' = 14 -> ( battery_low' = 1 & docked' = 1))

[SYS_LIVENESS]
## Goal Condition
cabinet1_surveyed' = 1 & cabinet2_surveyed' = 1 &
cabinet3_surveyed' = 1 & cabinet4_surveyed' = 1 &
cabinet5_surveyed' = 1 & docked' = 1

[ENV_TRANS]
## Action Effects
# moveToX effects: locationX is set to 1 and others are set to zero.
# All other variables remain unchanged
(action = 1) -> (at_cabinet1' = 1 & at_cabinet2' = 0
               & at_cabinet3' = 0 & at_cabinet4' = 0
               & at_cabinet5' = 0 & at_dockingArea' = 0
               & cabinet1_surveyed' = cabinet1_surveyed
               & cabinet2_surveyed' = cabinet2_surveyed
               & cabinet3_surveyed' = cabinet3_surveyed
               & cabinet4_surveyed' = cabinet4_surveyed
               & cabinet5_surveyed' = cabinet5_surveyed
               & docked' = docked & battery_low' = battery_low)
.............................................
.............................................

# survey_cabinetX effects: cainetX_surveyed is set to 1.
# All other variables remain unchanged
(action = 7) -> (.....................................
```

```
                & cabinet1_surveyed' = 1.................)
.................................................
.................................................

# unDock effects: docked is set to 0.
# All other variables remain unchanged
(action = 12) -> (.....................................
                  & docked' = 0 & ......................)

# dock effects: docked is set to 1.
# All other variables remain unchanged
(action = 13) -> (.....................................
                  & docked' = 1 & ......................)

# chargeBattery effects: battery_low is set to 0.
# All other variables remain unchanged
(action = 14) -> (.....................................
                  & battery_low' = 0 & .................)
```

Table 4.2: Sample run of the robot in mock-up vault

Atom	1	2	3	4	5	6	7	8	9	10	11	12	13	14
docked	1	0	0	0	0	0	0	0	0	0	0	0	0	1
at_dockingArea	1	1	0	0	0	0	0	0	0	0	0	0	1	1
battery_low	0	0	0	0	0	0	0	0	0	0	0	0	0	0
at_cabinet1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
at_cabinet2	0	0	0	0	1	1	0	0	0	0	0	0	0	0
at_cabinet3	0	0	0	0	0	0	1	1	0	0	0	0	0	0
at_cabinet4	0	0	0	0	0	0	0	0	1	1	0	0	0	0
at_cabinet5	0	0	0	0	0	0	0	0	0	0	1	1	0	0
cabinet1_surveyed	0	0	0	1	1	1	1	1	1	1	1	1	1	1
cabinet2_surveyed	0	0	0	0	0	1	1	1	1	1	1	1	1	1
cabinet3_surveyed	0	0	0	0	0	0	0	1	1	1	1	1	1	1
cabinet4_surveyed	0	0	0	0	0	0	0	0	0	1	1	1	1	1
cabinet5_surveyed	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Action	12	1	7	2	8	3	9	4	10	5	11	6	13	0

Refer to the Table 4.2 for a sample run of the inventory inspection. The extracted strategy is the same as we found in GOAP i.e. undock, visit

each cabinet, survey each cabinet, get back to the docking area and re-dock. This example establishes the Corollary 3.1.2. It should be noted that the order in which the shelves are visited isn't fixed and can be shuffled as well. This system is also compatible with existing higher level control loops present for collision and radiation detection.

Table 4.3: Sample run of the robot in mock-up vault with interrupt

Atom	7	8	9	10	11	12	13	14	15	16	17	18	19
docked	0	0	1	1	0	0	0	0	0	0	0	0	1
at_dockingArea	0	1	1	1	1	0	0	0	0	0	0	1	1
battery_low	1	1	1	0	0	0	0	0	0	0	0	0	0
at_cabinet1	0	0	0	0	0	0	0	0	0	0	0	0	0
at_cabinet2	0	0	0	0	0	0	0	0	0	0	0	0	0
at_cabinet3	1	0	0	0	0	1	1	0	0	0	0	0	0
at_cabinet4	0	0	0	0	0	0	0	1	1	0	0	0	0
at_cabinet5	0	0	0	0	0	0	0	0	0	1	1	0	0
cabinet1_surveyed	1	1	1	1	1	1	1	1	1	1	1	1	1
cabinet2_surveyed	1	1	1	0	0	1	1	1	1	1	1	1	1
cabinet3_surveyed	0	0	0	0	0	0	1	1	1	1	1	1	1
cabinet4_surveyed	0	0	0	0	0	0	0	0	1	1	1	1	1
cabinet5_surveyed	0	0	0	0	0	0	0	0	0	0	1	1	1
Action	6	13	14	12	3	9	4	10	5	11	6	13	0

In the above run, we are assuming battery_low to be set to 0 i.e. the battery is never low throughout the run. The low battery can be used as an interrupt condition. We aren't modeling the interrupt conditions as a part of the planner but we will consider its implementation similar to the one presented in [18] i.e. interrupt will modify the world-state and re-call the planner to create a new plan which accounts for the condition.

Consider a run which runs properly till time = 7 in Table 4.2. It has

already surveyed the cabinet1 and cabinet2 and is at cabinet3. But before taking the action at this temporal phase, battery_low is flagged high. The system will stop and re-call the planner with the modified initial condition as highlighted below. It should be noted that everything else remains same in the Slugs encoding except the initial state.

```
.............
[ENV_INIT]
## Re-initialization of environment variables
docked = 0
at_dockingArea = 0
battery_low = 1
cabinet1_surveyed = 1
cabinet2_surveyed = 1
cabinet3_surveyed = 0
cabinet4_surveyed = 0
cabinet5_surveyed = 0
at_cabinet1 = 0
at_cabinet2 = 0
at_cabinet3 = 1
at_cabinet4 = 0
at_cabinet5 = 0
.............
```

Table 4.3 shows the sample run for interrupt condition. It moves back to the docking area, dock itself, and charge the battery. After charging the battery, it undocks and continues surveying the remaining cabinets before moving back to docking area again.

4.2 Robot Navigation in a dynamic environment

In example 4.1, we considered a deterministic system. All the actions were assumed to be perfect and the environment was static with only one agent operating in it. Let's increase the level of abstraction in this example

and remove the assumption on the environment being static, and then compare the performance of reactive synthesis with GOAP for the same example.

Consider the setup shown in Figure 4.4. This can be seen as a representative of a collaborative environment where the humans and the robots share the same work-space. The robot and the humans have their own assigned jobs. For example, the robot needs to go and inspect each fire-hose for leakages (highlighted by green crosses). In this example, we will assume the humans to move non-deterministically in the environment as we want the robot to take care of the errors that the humans can create.

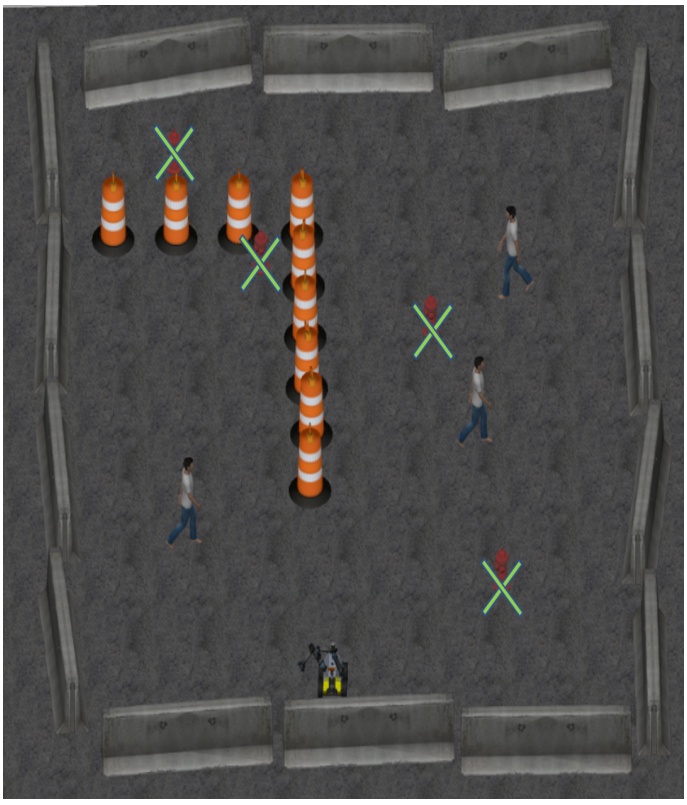


Figure 4.4: Gazebo view of a setup with dynamic external agents

The robot shown in the model 4.4 is NRG Vaultbot as shown in figure 4.5. It is built on a Clearpath Husky base, which is a high payload, four wheeled, skid-steered platform. A pair of Universal Robots UR5 manipulators are mounted on top of the Husky. The UR5 is a 6-Degree of Freedom (DOF) arm with a 5 kg payload. This is sufficient for mounting grippers and handling small objects. A Lidar system is equipped on the front of the base for 3D vision. For manipulation tasks, the UR5s are fitted with Robotiq 3-Finger Grippers

Figure 4.5: NRG Vaultbot

Figure 4.6 shows the abstracted form of the problem where the continuous world has been mapped to a grid world. At each location, the robot can move to the adjacent location by one step. The robot starts from the red block $(x = 4, y = 1)$ and needs to visit all the green blocks. Apart from our robot, there are three other dynamic obstacles (humans) operating in the environment moving non-deterministically. y-coordinates of all the obstacles are fixed but can move in the x-direction. There are also fixed obstacles (construction barrels) and walls shown by grey color.

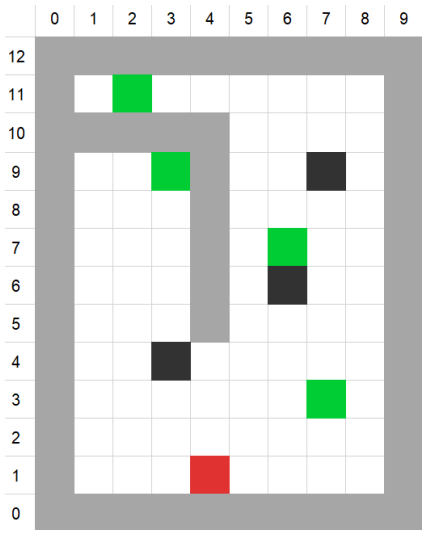


Figure 4.6: Grid world for robot navigation

We want the robot to perform some tasks at the green marked blocks while minimizing the number of steps moved by the robot. At the same time, we want to dodge all the obstacles and never collide with the wall.

4.2.1 ADLnE formulation

ADLnE can be used to model the environment as well as the controller. We represent the position of the robot by a_x and a_y. Similarly, we are using $o1\_x$, $o2\_x$ and $o1\_y$ etc to represent the locations of the obstacles. The controller has 8 choices at each location i.e. it can move 1 step in any direction or can perform the task depending on the current location of the robot. A 6-bit variable step is used to keep track of the number of steps the robot is taking to achieve the goal. As reactive synthesis doesn't have any concept of optimality attached to it, we will use this variable to calculate the (sub-)optimal solution

Action(moveNorth, SYS_1,

PRECOND: TRUE

EFFECT: {(a_y = a_y + 1), (a_x = a_x), (step = step + 1)})

Action(moveWest, SYS_1,

PRECOND: TRUE

EFFECT: {(a_y = a_y), (a_x = a_x - 1), (step = step + 1)})

Each task can be further modeled as a separate planning problem. We are not explicitly modeling the tasks in this example and tasks have been abstracted as a single temporal event. Statuses of the tasks are recorded by the Boolean variables taskXDone for all $X \in \{1, 2, 3, 4\}$

Action(doTask1, SYS_1,

PRECOND: {(a_y = 3), (a_x = 7)})

EFFECT: {(task1Done = 1)})

Safety conditions include avoiding the moving obstacles and remaining inside the world at all times. These conditions can be encoded as the safety conditions being controlled by different system agent_tags as follows:

Action(noWallCollision1, SYS_2,

PRECOND: TRUE

EFFECT: { ¬ (a_x = 1), (a_y = 2) })

Action(AvoidObstacle1, SYS_3,

PRECOND: TRUE

EFFECT: { ¬ ((a_x = o1_x), (a_y = o1_y)) })

Action(AvoidObstacle2, SYS_4,

PRECOND: TRUE

EFFECT: { ¬ ((a_x = o2_x), (a_y = o2_y)) })

Moving obstacles are modeled as moving non-deterministically in the environment. The y-coordinates of the obstacles are assumed to be fixed and the x-coordinates are non-deterministically modeled. The obstacle can stay at it's original position or move one step in x-direction without hitting the walls.

Action(moveObstacle1_1, ENV_1,

PRECOND: { (o1_x = 2) }

EFFECT: { {(o1_x = 1)}, {(o1_x = 3)} })

Action(moveObstacle1_2, ENV_1,

PRECOND: { {(o1_x = 1)}, {(o1_x = 3)} }

EFFECT: {(o1_x = 2)})

By using the translation laws defined in Section 3.2, we can convert these definitions to LTL and formulate it as a reactive synthesis problem. Following is the snippet of the Slugs encoding:

```
[INPUT]
## Variables to encode obstacle positions
o1_y:8...8
o1_x:1...3
o2_y:6
..........

## Variables to encode robot's positions
a_x: 0...9
a_y: 0...12
```

```
## Booleans to keep track of the status of tasks
task1Done:0...1
task2Done:0...1
task3Done:0...1
task4Done:0...1

## Variable to track the number of steps
step:0...50

[OUTPUT]
#0 = no motion, 1 = North, 2 = East, 3 = South, 4 = West
#5 = doTask1, 6 = doTask2, 7 = doTask3, 8 = doTask4
action:0...8

[ENV_INIT]
## Initialize the robot's position
## We don't need to initialize the obstacles coordinates
a_x = 4
a_y = 11

## Tasks are not done
task1Done = 0
.............

## No steps taken at the start
step = 0

[SYS_TRANS]
## Don't move out of the world
(a_x' != 0) & (a_y' != 0) & (a_x' != 9) & (a_y' != 12)

## Don't collide with the wall
!(a_x' = 1 & a_y' = 2)
!(a_x' = 2 & a_y' = 2)
.......................
!(a_x' = 4 & a_y' = 7)

## Avoid the moving obstacles
!(a_x' = o1_x & a_y' = o1_y)
............................
!(a_x' = o3_x' & a_y' = o3_y')

## Pre-conditions for doing the tasks
(action' = 5) -> (a_x' = 2 & a_y' = 1)
(action' = 6) -> (a_x' = 3 & a_y' = 3)
(action' = 7) -> (a_x' = 6 & a_y' = 5)
(action' = 8) -> (a_x' = 7 & a_y' = 9)

[SYS_LIVENESS]
```

```
## Goal COnditions
(task1Done' = 1) & (task2Done' = 1) &
(task3Done' = 1) & (task4Done' = 1)

[ENV_TRANS]
## Effect of motion
(action = 1) -> ((a_y' = a_y + 1) & (a_x' = a_x) & (step' = step + 1)
               & (task1Done' = task1Done) & (task2Done' = task2Done)
               & (task3Done' = task3Done) & (task4Done' = task4Done))
.................

## Effect of doing a task
(action = 5) -> ((a_y' = a_y) & (a_x' = a_x) & (step' = step)
               & (task1Done' = 1) & (task2Done' = task2Done)
               & (task3Done' = task3Done) & (task4Done' = task4Done))
.................

## Obstacles motion
(o1_x = 2) -> ((o1_x' = 1) | (o1_x' = 3))
((o1_x = 1) | (o1_x = 3)) -> (o1_x' = 2)
.......................
((o3_x = 5) | (o3_x = 7)) -> (o3_x' = 6)

[ENV_LIVENESS]
## Environment fairness property
## Without this external agent can pose deadlock
## and a strategy can't be found
(o1_x' = 1)
..........
```

4.2.2 GOAP formulation

As the GOAP framework can only handle the deterministic actions, and its ROS package [28] represents the states as the Booleans, we can't directly translate the system actions from ADLnE. We overcome this limitation by grounding the actions [51] to formulate it in GOAP. For example, to move to coordinate $(5, 3)$ in one step, we can approach it from any direction. Four possible motions are moveSouthFrom_x_5_y_4, moveNorthFrom_x_5_y_2, moveWestFrom_x_6_y_3, moveEastFrom_x_4_y_3. Following are some of the

action definitions in GOAP:

Action(moveSouthFrom_x_5_y_4,
PRECOND: { At(x_5,y_4) },
EFFECT: { At(x_5,y_3), ¬ At(x_5,y_4) })

Action(moveEastFrom_x_5_y_4,
PRECOND: { At(x_5,y_4), }
EFFECT: { At(x_6,y_4), ¬ At(x_5,y_4) })

Action(doTask1,
PRECOND: { At(x_2,y_1), }
EFFECT: { task1Done })

GOAP [28] also provides the methods to input the costs of the actions. We will use this to try to avoid the collisions. It should be noted that although the cost seems like a helpful tool here, it doesn't provide any guarantees about the safety of the robot. At best, if a location is occupied by the moving obstacle and planner wants to move to that location, an interrupt condition will be triggered to re-plan as we saw in the last example 4.1. As GOAP cannot model the dynamic obstacles, for the moving obstacles, we will use the constant cost for all the possible locations that the obstacle can reach at. We are defining two variables: movingObstacleCost = 30 and wallCollisionCost = 100 to give more weight to the wall collision. Following are a few examples of applying the costs to the grounded actions. Other actions can also be assigned costs in a similar way.

setCost(moveSouthFrom_x_3_y_11, 100)

setCost(moveEastFrom_x_3_y_6, 100)

setCost(moveNorthFrom_x_6_y_5, 30)

setCost(moveWestFrom_x_4_y_4, 30)

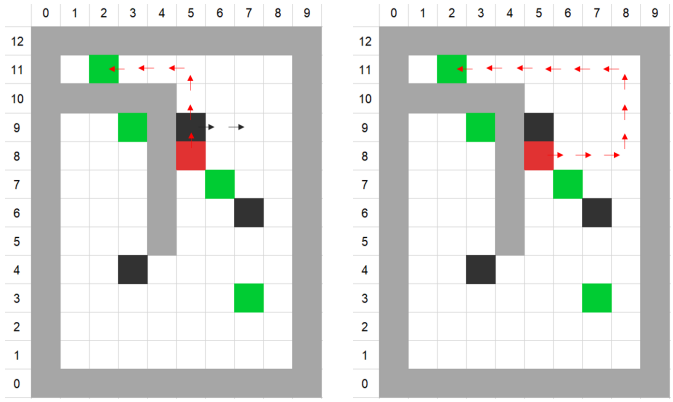
Table 4.4: Strategy comparison between ADLnE and GOAP for 10000 runs

Property	GOAP	ADLnE
Time to extract out the strategy	<1s	<1s
Number of Collisions	4099	0
Number of Interrupts	11364	0
Average number of spatial steps per run	38	33
Average number of time steps per run	42.13	57.36

The extracted strategies from ADLnE and GOAP were used to run the agent in the simulated world 4.6. Time to extract the strategy in both the cases was quite low as the problem size is really small. Even for a large state space, both the methods will be expected to perform similarly as both are exponentially complex. These strategies can be compared on the basis of safety, robustness, and the time taken by the robot to achieve all the goals. Table 4.4 summarizes some of the results for 10000 runs in the simulated world.

In GOAP, the interrupt condition is triggered if the action tries to move to a location which is already occupied by the obstacle. If the obstacle moves to the location occupied by the robot, then that condition is categorized as a collision. As expected, our formulation gives the safety guarantees and there are no collisions in any run. Also, as we are modeling the environment, there

is no need for interrupts as well. On the other hand for GOAP, there were 4099 collisions and 11364 interrupts in 10000 runs.



(a) ADLnE Sample Run (b) GOAP Sample Run

Figure 4.7: Sample run of the extracted strategies in dynamic environment

Additionally, the number of spatial steps needed to achieve the goals is 33 in ADLnE in comparison to 38 in GOAP. But it comes at the cost of the time. GOAP needs 42.13 time steps on an average but our strategy will need more than 57 time steps to achieve all the goals. Figure 4.7 shows a sample run for both ADLnE and GOAP. Time difference comes from the right-hand side of the world. In order to minimize the number of steps, the robot has to wait for more time to take the shortest route in the ADLnE strategy. For GOAP, the strategy decides to navigate away from the paths of the obstacles. This increases the spatial steps but decreases the time.

4.3 Switching Protocol

This example can be seen as a representative of a broad class of reactive switching protocol problems [52, 53, 54, 55]. The control modes in which the robot can operate can satisfy a set of specifications but are not assured to satisfy mission level specifications which makes it inevitable to switch between these modes. The objective of this example is to synthesize switching protocols that determine the sequence in which the modes of a switched system are activated to satisfy certain high-level specifications. Different modes may correspond to, for example, the evolution of the system under different, pre-designed feedback controllers or motion primitives in robot motion planning. In general, any system modeled with the following set of equations can be formulated similar to this example:

$$\dot{\mathbf{x}} = \mathbf{f_d}(\mathbf{u(t)}, \mathbf{e(t)}) \tag{4.1}$$

where $\mathbf{f_d}$ is a function over $\mathbf{u}$ and $\mathbf{e}$ and $d \in D$ denotes the set of modes in which our system can operate (e.g. the set of maneuvering directions of the robot). $\mathbf{u}$ is the control signal and $\mathbf{e}$ is the corresponding error.

Though not handled in this work, discrete states can be abstracted from the continuous dynamics by introducing an abstraction mapping between the continuous and the discrete domain and further can be formally defined as a transition system by over-approximation of the system [56]. Over-approximation takes into account all the possible transitions from a given initial state to all reachable states.

In example 4.2, we considered the navigation problem of a robot with perfect actions. In this example, we will remove that assumption and actions are modeled to be erroneous.

4.3.1 Problem Statement and ADLnE definitions

Consider the robot navigation problem in a non-deterministic environment. Figure 4.8 shows the 10×10 grid world that the robot is navigating in(from the start location (red block) to goal(green block)). The dotted region represents the wall. Dark blocks represent the external obstacles which are moving in the environment.

The robot receives 2 control signals: controller number ($p \in 0, 1, 2$) and direction ($d \in toStay, North, East, South, West$. For our analysis 3 controllers are considered such that $|u_0| < |u_1| < |u_2|$ and $|e_0| < |e_1| < |e_2|$. In our case, u_p refers to the number of cells that the robot can traverse in one time-step. Hence, the controller 2 can provide the highest speed but has poor accuracy as compared to other 2 controllers. Similarly, controller 1 can provide higher speed compared to controller 0 but is less accurate when compared to 0. More formally, it can be represented as

$$\dot{\mathbf{x}} = \mathbf{f_d}(\mathbf{u_p}(\mathbf{t}), \mathbf{e_p}(\mathbf{t})) \tag{4.2}$$

where $\mathbf{f_d}$ is the linear function of $\mathbf{u_p}$ and $\mathbf{e_p}$ and $d \in D$ denotes the set of maneuvering directions of the robot. $\mathbf{u_p}$ is the control signal and $\mathbf{e_p}$ is the

corresponding error associated with the p^{th} controller; $p \in P$ denotes the set of control actions.

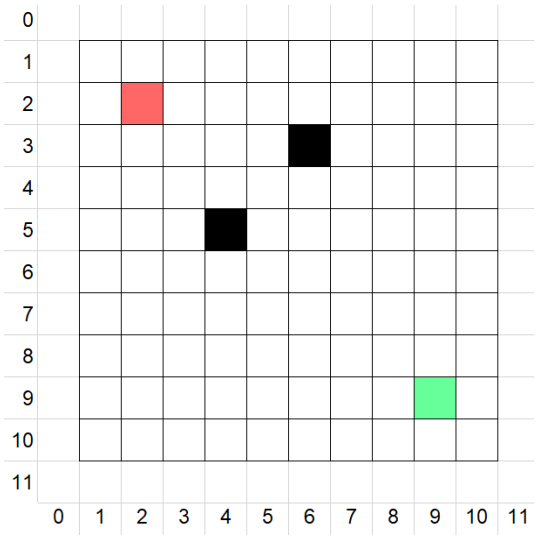


Figure 4.8: 10×10 grid world for robot navigation with erroneous actions

There are two control variables: *controller* and *direction*. *direction* can take value from 0 to 4 (0 = no motion, 1 = North, 2 = East, 3 = South, 4 = West) and *controller* gives the choice to the system to choose among the 3 controllers. Controller 0 refers to the controller with zero error and can move the robot to the adjacent cell depending on the *direction* $\in 1, 2, 3, 4$. Controller 1 can move the robot in the assigned direction to the 3^{rd} cell from its current location but will have an error of ± 1 in the perpendicular direction of motion. Similarly, controller 2 can move to the 4^{th} cell in the direction of motion and error of ± 2 in the perpendicular direction. Hence, there are 12 actions (4 directions and 3 controllers) that the robot can perform. $a\_x \in 0...11$

and $a\_y \in 0...11$ denote the coordinates of the robot. Example of some of the actions defined in ADLnE are as follows:

Action(north3Steps, SYS_1,
PRECOND: {(a_y $\leq$ 8), (a_x $\geq$ 1), (a_x $\leq$ 10)}
EFFECT: {(a_y = a_y+3), (a_x = a_x)}, {(a_y = a_y+3), (a_x = a_x+1)}, {(a_y = a_y + 3), (a_x = a_x-1)})

Action(west1Step, SYS_1,
PRECOND: TRUE,
EFFECT: {(a_y = a_y), (a_x = a_x - 1)})

Obstacles are modeled as moving non-deterministically in the environment. x-coordinates of the obstacles are fixed and 3-bit variables are used to capture the y-coordinate (o1_y $\in$ 4...10, o2_y $\in$ 1...7). There is a discrete battery variable *battery* $\in$ 0...31 which keeps track of the battery level and decreases by one level per time step.

Action(moveObstacle1, ENV_1,
PRECOND: TRUE,
EFFECT: {(o1_y = o1_y + 1)}, {(o1_y = o1_y)}, {(o1_y = o1_y - 1)})

Action(moveObstacle2, ENV_2,
PRECOND: TRUE,
EFFECT: {(o2_y = o2_y + 1)}, {(o2_y = o2_y)}, {(o2_y = o2_y - 1)})

We want to synthesize a planner such that the robot never collides with any obstacle or gets into the restricted region $((x = 0)|(y = 0)|(x =$

11)|($y = 11$)). Also, it should eventually get to its goal position $(9, 9)$. *battery* is fully charged (i.e. reset to 31) when the robot visits $(2, 2)$. We want to keep the battery level in good working conditions (i.e. $battery \geq b_{threshold}$). It's interesting to note that the battery reset condition and minimumBattery condition force the robot to get back to the home $(2, 2)$, even though we never explicitly set $(2, 2)$ as part of the mission.

Action(batteryStatus, ENV_3,
PRECOND: {¬((a_x = 2), (a_y = 2))},
EFFECT: {(battery = battery - 1)})

Action(batteryReset, ENV_3,
PRECOND: {(a_x = 2), (a_y = 2)},
EFFECT: {(battery = 31)})

Action(minimumBattery, SYS_2,
PRECOND: TRUE,
EFFECT: {(battery ≥ 8)})

Action(noWallCollision, SYS_3,
PRECOND: TRUE,
EFFECT: {(a_x != 0),(a_x != 11),(a_y != 0),(a_y != 11) })

Goal({(a_x = 9), (a_y = 9)})

By using the translation laws defined in Section 3.2, we can convert these definitions to LTL and formulate it as reactive synthesis. Further, we can extract the automaton which captures the states of the world. Following

is the Slugs [40] encoding for this example:

```
[INPUT]

## Variables for modeling of obstacles
o1_y:4...10
o2_y:1...7
# Constant x-coordinate
o1_x:4...4
o2_x:6...6

## Variables for modeling of robot position
a_x: 0...11
a_y: 0...11

## Battery status and action for transition in battery level
battery:0...31
batteryAction:0...1

[OUTPUT]

## Controller Actions Enumeration
# 0 = No motion
# mode0: 1 = North, 2 = East, 3 = South, 4 = West
# mode1: 5 = North, 6 = East, 7 = South, 8 = West
# mode2: 9 = North, 10 = East, 11 = South, 12 = West
action:0...12

[SYS_INIT]
## Not initializing the controller to any state

[ENV_INIT]

## Environmental variables initialization
o1_y = 7
o2_y = 4
a_x = 2
a_y = 2
batteryAction = 1
battery = 31

[SYS_TRANS]

## Safety Conditions

# No collision with wall
a_x' != 0
a_y' != 0
a_x' != 11
```

```
a_y' != 11

# Avoid the obstacle
!(a_x <= o1_x & a_x' >= o1_x & (a_y = o1_y | a_y' = o1_y
| a_y = o1_y'))
!(a_x >= o1_x & a_x' <= o1_x & (a_y = o1_y | a_y' = o1_y
| a_y = o1_y'))
!(a_x <= o2_x & a_x' >= o2_x & (a_y = o2_y | a_y' = o2_y
| a_y = o2_y'))
!(a_x >= o2_x & a_x' <= o2_x & (a_y = o2_y | a_y' = o2_y
| a_y = o2_y'))

# Maintain the good battery status
battery' >= 8

## Pre-conditions for the System Actions
(action' = 5 -> (a_y' <= 8 & a_x' >= 1 & a_x' <= 10))
(action' = 6 -> (a_x' <= 8 & a_y' >= 1 & a_y' <= 10))
(action' = 7 -> (a_y' >= 3 & a_x' >= 1 & a_x' <= 10))
(action' = 8 -> (a_x' >= 3 & a_y' >= 1 & a_y' <= 10))
(action' = 9  -> (a_y' <= 7 & a_x' >= 2 & a_x' <= 9))
(action' = 10 -> (a_x' <= 7 & a_y' >= 2 & a_y' <= 9))
(action' = 11 -> (a_y' >= 4 & a_x' >= 2 & a_x' <= 9))
(action' = 12 -> (a_x' >= 4 & a_y' >= 2 & a_y' <= 9))

[SYS_LIVENESS]

## Goal Condition
a_x' = 9 & a_y' = 9

[ENV_TRANS]

## Battery Action pre-conditions and effect
(batteryAction' = 0) -> !(a_x' = 2 & a_y' = 2)
(batteryAction' = 1) -> (a_x' = 2 & a_y' = 2)
(batteryAction = 0) -> battery' + 1 = battery
(batteryAction = 1) ->  battery' = 31

## Obstacle dynamics
(o1_y' + 1 = o1_y) | (o1_y' = o1_y) |(o1_y' = o1_y + 1)
(o2_y' + 1 = o2_y) | (o2_y' = o2_y) |(o1_y' = o2_y + 1)

## Action Effects
(action = 0) -> ((a_x' = a_x) & (a_y' = a_y))

# Single step actions without any error
(action = 1) -> ((a_y' = a_y + 1) & (a_x' = a_x))
(action = 2) -> ((a_x' = a_x + 1) & (a_y' = a_y))
(action = 3) -> ((a_y' + 1 = a_y) & (a_x' = a_x))
```

```
(action = 4) -> ((a_x' + 1 = a_x) & (a_y' = a_y))

# 3 step actions
(action = 5) -> ((a_y' = a_y + 3) & (a_x' + 1 >= a_x)
& (a_x + 1 >= a_x'))
(action = 6) -> ((a_x' = a_x + 3) & (a_y' + 1 >= a_y)
& (a_y + 1 >= a_y'))
(action = 7) -> ((a_y' + 3 = a_y) & (a_x' + 1 >= a_x)
& (a_x + 1 >= a_x'))
(action = 8) -> ((a_x' + 3 = a_x) & (a_y' + 1 >= a_y)
& (a_y + 1 >= a_y'))

# 4 step actions
(action = 9 ) -> ((a_y' = a_y + 4) & (a_x' + 2 >= a_x)
& (a_x + 2 >= a_x'))
(action = 10) -> ((a_x' = a_x + 4) & (a_y' + 2 >= a_y)
& (a_y + 2 >= a_y'))
(action = 11) -> ((a_y' + 4 = a_y) & (a_x' + 2 >= a_x)
& (a_x + 2 >= a_x'))
(action = 12) -> ((a_x' + 4 = a_x) & (a_y' + 2 >= a_y)
& (a_y + 2 >= a_y'))
```

Table 4.5 shows the number of states of the automaton extracted for different scenarios. It can be easily inferred that the size of the automaton of realizable strategy increases exponentially with the increase in the encoded states of the system. It requires 19105 states when both the obstacles are moving compared to 187 when there are no obstacles in the environment. Minimum time to satisfy all the specifications also increases with dynamic obstacles in the environment.

It is interesting to note the difference in the cases when one of the obstacles is fixed in contrast to the case when that obstacle is not present at all. For example, compare the case 5 and 7 where obstacle 1 is moving. The minimum time is the same for both the cases but when obstacle 2 is present(fixed), the size of the automaton is much smaller than the case when

Table 4.5: Automaton size comparison

Sr.no.	Obstacle 1	Obstacle 2	Minimum battery	Size of automaton
1	not present	not present	16	187
2	not present	fixed	16	192
3	fixed	not present	16	187
4	fixed	fixed	15	199
5	moving	fixed	15	1299
6	fixed	moving	14	1514
7	moving	not present	15	1745
8	not present	moving	14	2021
9	moving	moving	8	19105

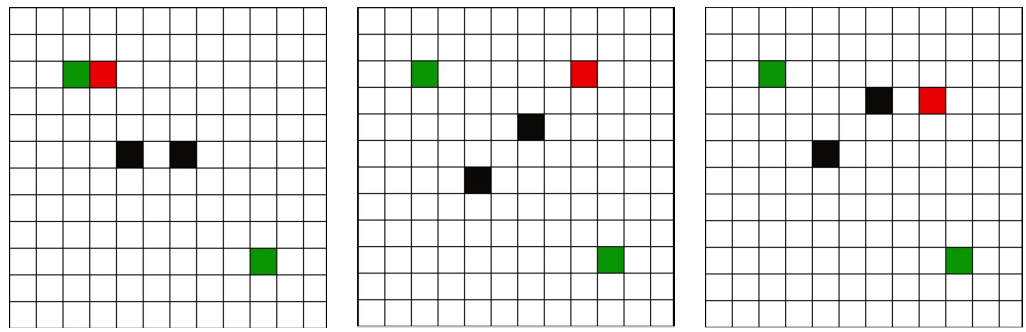
obstacle 2 is not present. Presence of obstacle 2 can be seen as restricting the allowed behavior of the system (controller and direction). In other words, all the states allowed in case 5 form the subset of the states allowed in case 7.

4.3.2 Sample Run

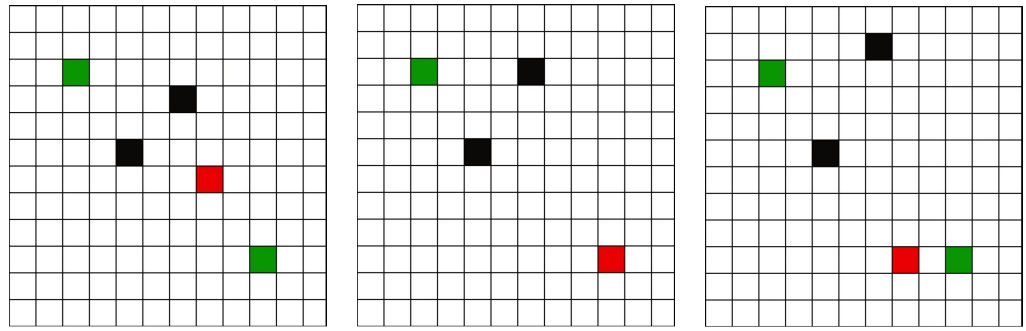
Figure 4.9 shows the states of the robot and the obstacles at various time steps. The robot is able to reach the goal state $(x = 9)$ & $(y = 9))$ without colliding with any obstacle or entering the restricted region $((x = 0)|(y = 0)|(x = 11)|(y = 11))$. Also, the robot gets back to home zone $(x = 2)$ & $(y = 2))$ to recharge the battery to maintain the minimum battery property.

A pair $Control = (direction, controller)$ can be used to represent the system decisions. As pointed out in Section 4.3.1, there are 3 controllers:

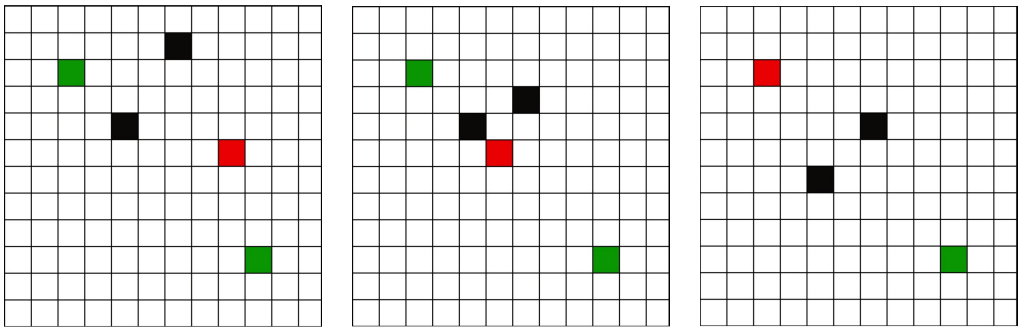
1. $controller = 0$: one step motion with zero error steps
2. $controller = 1$: three step motion with one perpendicular error steps
3. $controller = 2$: four step motion with two perpendicular error steps



(a) time = 1 (b) time = 6 (c) time = 7



(d) time = 8 (e) time = 11 (f) time = 14



(g) time = 15 (h) time = 18 (i) time = 20

Figure 4.9: Position of the robot and the obstacles at various time steps

At $time = 1$, robot leaves the start zone (2,2) by using $Control = (2, 0)$. By continuing for 5 more time steps in $Control = (2, 0)$ and taking first

$Control = (1, 0)$, the robot and the obstacles reach the state shown in Figure 4.9(c). At $time = 7$, system chooses $Control = (1, 1)$ that is move three steps in south but as the $controller = 1$ can have error of one step in east or west it ends up moving one step in west as well as shown in Figure 4.9(d). At $time = 11$, robot reaches the goal as shown in Figure 4.9(e). At $time = 14$, system chooses to move the robot by using $Control = (3, 2)$ as shown in Figure 4.9(f) to Figure 4.9(g) transition. As $controller = 2$ has highest error value associated with it, one would generally expect the system to never take this choice. But as can be seen at $time = 14$, no matter how the obstacles behave, choice of $controller = 2$ is still a safe choice (no collision possible) for the system. At $time = 20$, robot returns back to its starting zone maintaining the minimum battery specification.

Chapter 5

Conclusions and Future Work

5.1 Summary

In this work, we investigated the action planning for the high-level robot control. One of the key strengths of the action planners lies in the fact that they require only the capabilities of the robot, environment behavior and goal condition to do a task. From this, it calculates the sequence of actions that take the system to the goal state. Nevertheless, the classical action planners have some limitations restricting them for real-life applications. To alleviate some of these limitations, we presented the synthesis of *correct-by-construction* action planners. We have targeted the non-determinism, dynamic and multi-agent modeling. The preliminary results generated for this effort demonstrate their potential to act as an entry point for the planning community to employ formal methods to automate the synthesis of the action planners. In this work, we have proposed two aspects as initial steps towards that goal.

One is the improved action formalism (ADLnE) which targets the limitations of classical action languages. ADLnE provides more flexibility in the problem formulation. It can be used to model the multiple agents sharing the same environment/work-space and can also handle the non-deterministic

pre-conditions and effects for more realistic modeling. The uncontrolled environmental agents are assumed to be adversarial to our agent.

The second aspect is the translation of ADLnE to LTL specifications which can be formulated to a reactive synthesis problem. Reactive synthesis extracts the *action plan* as a finite automaton by modeling the problem as a two player game played between the robot and the environment. From its construction, the existence of a winning strategy for the system ensures that the system satisfies the given specifications (goal and safety conditions) against all the allowable behaviors of the environment. We implemented our approach for three different scenarios.

First, we established that the proposed action language (ADLnE) is at least as expressive as the classical action language, ADL which is also used in the formulation of GOAP. We tested this by formulating the previously studies vault surveying problem. All the actions were considered to be perfect and the environment was assumed to be static and deterministic. It was established that the extracted strategy is the same as that of GOAP.

Second, we formulated a navigation problem with the perfect actions but dynamic environmental agents. This example highlights the limitations of the classical action planners. The proposed approach performed significantly better when compared to the classical planners. With the classical planners strategies, the robot collided with the dynamic agents in approximately 40% of the runs and - on average - had to re-plan at least once in every run. On the other hand, our approach gives full proof of the safety conditions. There

were no collisions as the robot moves only if the environmental agent is at a safe distance. But this comes at the cost of more time taken by the robot to execute the synthesized strategy. The proposed planner took an average of 57 time steps in comparison to 42 in case of the GOAP. Additionally, it was found that the time to extract the strategy in both the cases was quite low. For a large state space, we expect both the methods to perform similarly in terms of scalability as both are exponentially complex.

Third, we considered the switching protocol problem. The individual control modes in which the robot can operate can satisfy a set of specifications but are not assured to satisfy mission level specifications which makes it inevitable to switch between these modes. As an exemplary, we formulated a navigation problem in a dynamic environment with erroneous actions possible from our robot. Through simulations, we showed that the robot was able to reach the goal state without colliding with the static as well as dynamic agents while remaining the safe battery level.

5.2 Recommendations for Future Work

There are a couple of interesting problems that need further exploration. In this work, the robot and obstacles are assumed to take only the discrete steps to perform an action. Further, we assume that the low-level routines can accurately implement the high-level command synthesized from our planner. For testing on real hardware, one approach can be implementing an task manager similar to Anderson et al. [18]. More robust approaches include the interfacing layers similar to He et al. [43] and Dantam et al. [57] to

establish a feedback mechanism between the high-level planner and the low-level implementations. This area of research is quite active and provides a promising solution for achieving more realistic robotic behaviors interacting with the dynamic environments.

We have also limited our discussion only to the goal reachability. We briefly talked about the optimality in Section 4.2, but a better definition and analysis of optimality remains unanswered. One approach can be to analyze the actions as the edges of a directed graph similar to the GOAP but with the unsafe actions pruned from the graph. This weighted directed graph can be used to further build the sub-optimal runs.

Finally, returning to the motivating application for performing inspection tasks at user designated locations in an operational oil rig platform, some implementations not discussed in this work include the assumptions of discrete space definitions and full observability. Nevertheless, our approach doesn't inject any implementation obstacles that cannot be addressed.

Reactive synthesis provides a deep insight into the worst-case analysis. Extending this work to handle probabilistic correctness guarantees and planner synthesis will be a natural step towards better action planning formulation. Incorporating probabilistic models, such as Markov decision process (MDP) [58] or Partially Observable Markov Decision Process (POMDP) [59] will be really helpful. Further, the POMDPs will help in relaxing the observability limitation highlighted in the Section 2.4 and can provide more realistic problem formulation.

Bibliography

[1] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[2] Michael Gelfond and Vladimir Lifschitz. Action languages. 1998.

[3] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[4] Edwin PD Pednault. Adl and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512, 1994.

[5] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[6] Stephen Kangogo Cherutich. Rig selection and comparison of top drive and rotary table drive systems for a cost effective drilling project. *Report*, 8:65–84, 2009.

[7] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[8] Alfonso E Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[9] Rajeev Alur, Thomas A Henzinger, Gerardo Lafferriere, and George J Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[10] Amit Bhatia, Matthew R Maly, Lydia E Kavraki, and Moshe Y Vardi. Motion planning with complex goals. *IEEE Robotics & Automation Magazine*, 18(3):55–64, 2011.

[11] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3116–3121. IEEE, 2007.

[12] Hadas Kress-Gazit, Tichakorn Wongpiromsarn, and Ufuk Topcu. Correct, reactive, high-level robot control. *IEEE Robotics & Automation Magazine*, 18(3):65–74, 2011.

[13] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297, 2008.

[14] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon control for temporal logic specifications. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 101–110. ACM, 2010.

[15] Scott C Livingston, Richard M Murray, and Joel W Burdick. Backtracking temporal logic synthesis for uncertain environments. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5163–5170. IEEE, 2012.

[16] Eric M Wolff, Ufuk Topcu, and Richard M Murray. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4332–4339. IEEE, 2013.

[17] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[18] Blake Anderson, Meredith Pitsch, Selma Wanna, David Park, Sheldon Landsberger, and Mitch Pryor. Autonomous inventory in nuclear environment using a remote platform. In *D and RS 2016 - Decommissioning and Remote Systems*, pages 103–107. American Nuclear Society, 1 2016.

[19] Edwin PD Pednault. Adl: Exploring the middle ground between strips and the situation calculus. *Kr*, 89:324–332, 1989.

[20] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22 (3):57, 2001.

[21] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[22] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[23] Jonathan Bohren and Steve Cousins. The smach high-level executive [ros news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[24] Christopher G Atkeson, BPW Babu, N Banerjee, D Berenson, CP Bove, X Cui, M DeDonato, R Du, S Feng, P Franklin, et al. What happened at the darpa robotics challenge, and why. *submitted to the DRC Finals Special Issue of the Journal of Field Robotics*, 1, 2016.

[25] Jeff Orkin. Symbolic representation of game world state: Toward real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, volume 5, pages 26–30, 2004.

[26] Jeff Orkin. Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006, page 4, 2006.

[27] Edmund Long. Enhanced npc behaviour using goal oriented action planning. *Master's Thesis, School of Computing and Advanced Technologies, University of Abertay Dundee, Dundee, UK*, 2007.

[28] Robert Blake Anderson. Github: task_planning. https://github.com/ utnuclearroboticspublic/task_planning, 2016.

[29] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

[30] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

[31] Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170(3):298, 2006.

[32] Hidetomo Nabeshima, Takehide Soh, Katsumi Inoue, and Koji Iwanuma. Lemma reusing for sat based planning and scheduling. In *ICAPS*, pages 103–113, 2006.

[33] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. *KR*, 96:374–384, 1996.

[34] Bernd Finkbeiner and Felix Klein. Reactive synthesis: Towards output-sensitive algorithms. *Dependable Software Systems Engineering*, 50:25, 2017.

[35] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[36] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[37] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the gap between fair simulation and trace inclusion. In *International Conference on Computer Aided Verification*, pages 381–393. Springer, 2003.

[38] Sebastian Sardina and Nicolás D'Ippolito. Towards fully observable non-deterministic planning as assumption-based automatic synthesis. In *IJCAI*, pages 3200–3206, 2015.

[39] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 313–314. ACM, 2011.

[40] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible gr (1) synthesis. In *International Conference on Computer Aided Verification*, pages 333–339. Springer, 2016.

[41] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy–a new requirements analysis tool with synthesis. In *International Conference on Computer Aided Verification*, pages 425–429. Springer, 2010.

[42] Ebru Aydin Gol, Mircea Lazar, and Calin Belta. Language-guided controller synthesis for linear systems. *IEEE Transactions on Automatic Control*, 59(5):1163–1176, 2014.

[43] Keliang He, Morteza Lahijanian, Lydia E Kavraki, and Moshe Y Vardi. Towards manipulation planning with temporal logic specifications. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 346–352. IEEE, 2015.

[44] Morteza Lahijanian, Shaull Almagor, Dror Fried, Lydia E Kavraki, and Moshe Y Vardi. This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction. In *AAAI*, pages 3664–3671, 2015.

[45] Amir Pneuli. The runner-blocker example, controller synthesis. http://www.dis.uniroma1.it/~degiacom/didattica/dottorato-amir-pnueli/slides/part2-4up.pdf, 2005.

[46] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell Marcus, and Hadas Kress-Gazit. Provably correct reactive control from natural language. *Autonomous Robots*, 38(1):89–105, 2015.

[47] Gangyuan Jing, Cameron Finucane, Vasumathi Raman, and Hadas Kress-Gazit. Correct high-level robot control from structured english. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3543–3544. IEEE, 2012.

[48] Ioannis Filippidis, Richard M Murray, and Gerard J Holzmann. A multi-paradigm language for reactive synthesis. *arXiv preprint arXiv:1602.01173*, 2016.

[49] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[50] Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

[51] Silvia Coradeschi, Amy Loutfi, and Britta Wrede. A short review of symbol grounding in robotic and intelligent systems. *KI-Künstliche Intelligenz*, 27(2):129–136, 2013.

[52] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[53] Ji-Woong Lee and Geir E Dullerud. Joint synthesis of switching and feedback for linear systems in discrete time. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, pages 201–210. ACM, 2011.

[54] Daniel Liberzon and A Stephen Morse. Basic problems in stability and design of switched systems. *IEEE Control systems*, 19(5):59–70, 1999.

[55] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 22–31. ACM, 2010.

[56] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Transactions on Automatic Control*, 58(7):1771–1785, 2013.

[57] Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and Systems*, pages 1–6, 2016.

[58] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, volume 1, pages 690–700, 2001.

[59] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.