Integrating a Planner Under a Platform Independent Software Architecture for Path-Planning in Mobile Robotics

Author:
Jonathan Ferrer Mestres

Supervisor:
Vladimir Estivill-Castro

A thesis submitted in fulfilment of the requirements for the master of Intelligent Interactive Systems

in the

Artificial Intelligence Group
Department of Technology Information and Communication

September 2013
Declaration of Authorship

I, Jonathan Ferrer Mestres, declare that this thesis titled, 'Integrating a Planner under a Platform Independent Software Architecture for Path-Planning in Mobile Robotics' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others. Specially, I have the collaboration and help of my supervisor and also the collaboration and assistance of the Machine Intelligence and Pattern Analysis Laboratory, in Griffith University, Australia, where they allowed me to work with their tools.

Signed: Jonathan Ferrer Mestres

Date: August 2013
“We can only see a short distance ahead, but we can see plenty there that needs to be done.”

Alan Turing
I describe a software architecture based on logic-based finite-state machines and a distributed whiteboard that enables the inclusion of many different planners into the capabilities of a mobile robot. The finite-state machines enables a high level description of the behavior in accordance with model-driven development. The machine itself can be formally verified by model-checking techniques. The different planners can be utilized with an application programming interface based on the distributed whiteboard architecture. Because all infrastructure is platform independent, the actual robots utilized could be differential robots, humanoid-robots, or even a simulator. The standardization of planning problems by their descriptions in PDDL has resulted in clear benchmarking of planners, and thus, in significant advances in reliable and efficient planning packages. These packages design plans as sequence of actions of the controllable robots in the environment. I show here that, provided that the adversaries follow a deterministic behavior, PDDL-planers can also be used in dynamic environments where uncontrollable adversaries may obstruct paths at some time in the future. Therefore, this environments can be used by mobile robots without the need to use more sophisticated planers where environments are modeled by MDPs. I created a planning API for integrating any PDDL-solver and use it to devise platform independent planning behavior. The API also have the ability of switching between PDDL-solvers or to change the integration cycle of the planner. My supervisor and me show that these two features are essential for the dynamic environments considered here.
Acknowledgements

Thanks to Dr. Vladimir Estivill-Castro, for his teaching and passion for robotics. Thanks to Dr. Hector Geffner, for his guidance and his help. Thanks to the Artificial Intelligence Group and MiPal team. Thanks to my family, for their patience and unconditional support...
## Contents

Declaration of Authorship ........................................ i

Abstract ...................................................................... iii

Acknowledgements ...................................................... iv

List of Figures ............................................................ vii

Abbreviations ............................................................. viii

1 Introduction ............................................................ 1
  1.1 Context ............................................................. 1
  1.2 Motivation .......................................................... 3
  1.3 Objective ............................................................ 4

2 Automated Planning ................................................... 5
  2.1 Planning: General description ............................... 5
     2.1.1 STRIPS ....................................................... 5

3 Logic-Based Finite-State Machines ............................... 7
  3.1 LLFSM Description .............................................. 7

4 Incorporating Planning into the FSMs ......................... 11
  4.1 FSM plan control ................................................. 11
     4.1.1 The “plan, execute, re-plan” cycle .................. 11
     4.1.2 Planning module API .................................... 12
     4.1.3 The inclusion of policies ............................... 14
     4.1.4 The “execution-driven” cycle ......................... 15

5 Model-Checking Behaviors .......................................... 17
  5.1 Formal verification ............................................... 17

6 Adversarial Dynamic Environments ................................ 19
  6.1 Deterministic Planning in Dynamic Environments ....... 19
     6.1.1 Illustration ................................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Planning API for Dynamic Environments</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Experiments and Results</td>
<td>27</td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental evaluation</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Related Work</td>
<td>29</td>
</tr>
<tr>
<td>8.1</td>
<td>Contrast</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions and Future Work</td>
<td>31</td>
</tr>
<tr>
<td>9.1</td>
<td>Conclusions</td>
<td>31</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>Bibliography</td>
<td>33</td>
</tr>
</tbody>
</table>
## List of Figures

3.1 Section of the vision pipeline to recognize a blob of orange as a ball. ...... 9
3.2 Theory that defines when a blob is a ball. ......................................... 9
4.1 The logic-labeled interpreted (and also compiled) used to run the plan, execute, re-plan cycle in our experiments. .......................... 14
5.1 NUSMV coding of the property that re-planning is certainly requested when an unknown obstacle is detected and the robot has stopped moving. 18
6.1 A that encodes the behavior of a simple adversary that loops in 4 positions. 21
6.2 File that provides the PDDL domain description for the adversary ....... 21
6.3 Environment for an illustrative instance of a dynamic environment. .... 24
6.4 An environment where deterministic planning with obstacle sensing and re-planning fails. ................................................................. 26
9.1 The agent can calculate with certain probability the adversarial behavior. 32
Abbreviations

PDDL  Planning Domain Definition Language
STRIPS  STanford Research Institute Problem Solver
MDP  Markov Decision Process
POMDP  Partially Observable Markov Decision Process
MDD  Model Driven Development
FSM  Finite State Machine
DPL  Decisive Plausible Logic
LLFSM  Logic Labeled Finite State Machine
UDP  User Datagram Protocol
CTL  Computation Tree Logic
LTL  Linear Temporal Logic
ROS  Robot Operating System
PLEXIL  PLan EXecution Interchange Language
PRS  Procedural Reasoning System
Dedicated to Eva and my family.
Chapter 1

Introduction

1.1 Context

Since the early merging of automation and artificial intelligence into robotics, planning has become an integral part of the capabilities one expects on a robot. Currently, one of the most accepted definitions of a robot is that "a robot is an autonomous system which exists in the physical world, can sense its environment and can act on it to achieve goals" [Mataric, 2007]. Those familiar with planning will recognize the references to actions and goals. However, one of the biggest challenges of robotics control has been the design, description and implementation of a software architecture that enables the integration of planning with the more and more sophisticated behaviors we expect in mobile robots that act in unpredictable environments.

Certainly, planning was not part of the early machines and automata that evolved from feedback-control and open-loop control (in fact, we do not use the word robot for those, like the controller in an air-conditioner). But certainly, the early software architectures of robotics were based on declarative, deliberative, symbolic, logic-based approaches that naturally included planning [Siegwart et al., 2011]. The most notable example of this was Shakey which used a STRIPS planner [Fikes and Nilsson, 1971]. The planning involved in deliberative architectures was blamed for slow performance issues, and it was regarded as brittle if only imprecise information was around. These difficulties faced by deliberative software architectures resulted in the emergence of reactive software architectures. The most well know approach here is the subsumption architecture [Brooks, 1991]. There are many examples of robots based on reactive systems that could do significant navigation while essentially structured as a reactive system [Mataric, 1992]. Nevertheless, hybrid software architectures for robotic systems have attempted to bring together the advantages of deliberative control and reactive control. Unfortunately, the
need emerges to be able to describe and integrate several intelligent behaviors in more sophisticated robotic applications. Today, one could argue that many mobile robotic applications are behavior-based software architectures [Arkin, 1998] as the environment is dynamic, noisy and unpredictable, the robot faces incomplete information and must be responsive. Since behavior-based control suggest one can integrate several concurrent behaviors that collaborate and cooperate in processing the information of sensors, it is natural that a planner should be part of the capacities of such system.

What it is addressed here is how to integrate such a planner in a software architecture that can enable all the earlier approaches of robotic control. For this, a software architecture based on finite-state machines whose transitions are labeled by queries to a deliberative system is proposed by the Machine Intelligence and Pattern Analysis Laboratory (MIPAL), in Griffith University, Australia. Moreover, these machines are used to drive through a generic interface of a generic planner that can be incorporated into the arrangement of behaviors. In addition, it’s important to aim for a standard integration and for this we consider using “classical planners”. Standardization is important because STRIPS-like planning [LaValle, 2006, page 49] is perhaps the fundamental form of planning, and in particular, rather common planning for navigation tasks. While the robotics community has moved forward into problems that include temporal environments with uncontrollable obstacles, the planning community has standardized STRIPS-like planning with the introduction of the Planning Domain Definition Language (PDDL) and its extensions [Fox and Long, 2003; 2006 in order to consider mixed discrete-continuous domains.

Moreover, while the original deterministic and observable planning challenges have now been replaced by planning and learning domains as well as uncertainty challenges (where the effects of the actions are uncertain and usually modeled by some probability distribution of the outcomes), the current mainstream approach (to model partial observability as non-deterministic search [Bonet and Geffner, 2000] for both a) contingent planning [Hoffmann and Brafman, 2006] and b) POMDP planning [Kaelbling et al., 1998], has shown serious scalability issues [Bonet and Geffner, 2011]. Therefore, several researchers have studied classes of problems and produced methods to translate the partial observability problem description into a fully observable planning problem solved by a classical planner [Bonet and Geffner, 2011]. Moreover, I aim for a standard integration and for this I consider using “classical planners” using PDDL and its extensions.

PDDL has supported the AIPS Planning Competition in several editions, this has resulted in a series of benchmark planning problems (for example, see http://www.plg.inf.uc3m.es/ipc2011-deterministic/Domains). It also has resulted in an expansion of
the challenges. The original fully deterministic and observable planning challenges have been complemented by planning and learning domains as well as uncertainty challenges (where the effects of the actions are uncertain and usually modeled by some probability distribution of the outcomes). Nevertheless, I am motivated by the fact that the domains under the original (or “classical”) planning part, seem mostly derived from scheduling challenges. This is not surprising, as the plan is usually a schedule of the actions of the agent/robot in the domain. If the domain contains more controllable agents, they are also specified as part of the actions in a sequence. A prototype of this type of challenges is ‘Sokoban’ [Junghanns and Schaeffer, 2001]. In this transport puzzle, the agent/robot pushes crates (or boxes) in the interior of a warehouse. The goal is to place all objects in special cells of the grid environment that are designated as storage locations. The agent is confined to the warehouse, and actions are horizontally moves or vertically moves onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes nor into walls, and they cannot be pulled.

The second part of the challenge I pose to this type of problems is to truly consider them in a robotic environment where planning may happen along carrying out a particular plan (which may imply re-planning), and also, in the presence of other robots for which the robot has no control. These adversaries will perform deterministic trajectories governed by logic-labeled finite-state machines (llfsm). Therefore, this paper proposes (1) a methodology to describe (in PDDL) scenarios where the controllable robot is not the only moving objects in the environment. It also indicates (2) a software architecture for carrying out model-driven development of the integration of the planners into the capacities of a robot. These proposals are demonstrated with a simple scenario of a robot navigating in an environment with fixed obstacles as well as roving adversaries.

1.2 Motivation

If we rely at this time, we can ask what is the special interest about this work? Well, first of all, I considerer the interest to represent the planning problems of artificial intelligence in a humanoid robotic platform, i.e. transfer computer simulations to a robotic platform that interacts directly with the environment and acts in the real world. Furthermore, the implementation of a planning module, allows to do and solve, different types of problems using artificial intelligence algorithms on the different robots, not only robot navigation, but also robot interaction and robot acting with the environment, with different objects and obstacles moving around without any control done by the agent.
1.3 Objective

The main objectives of this master thesis are:

- To integrate Classical Planners on a robot and evaluate them in a probabilistic world.

- To extend a Planning Module that allows to integrate several planners under a Platform Independent Software.

- To develop efficient Path Finding for mobile robots in dynamic environments with adversaries.
Chapter 2

Automated Planning

2.1 Planning: General description

Automated Planning solves a problem in a space states using a sequence of actions. These sequences of actions have preconditions that have to be satisfied, and the use of these actions on the environment produces effects. These actions are performed by an agent and in this case, by a robot on the real world. The basic definition for classical planning problems is:

- A state space $S$.
- An initial state space $s_0 \in S$.
- A goal state $s_G \in S$.
- A set of actions $A(s)$ for each $s \in S$.
- A transition function $s' = f(s, a)$ where $a \in A(s)$.
- A function cost $c(a s)$ where $a \in A(s)$ and $s \in S$.

A solution for a planning problem is a sequence of actions that maps $s_0$ into $s_G$. The cost used on the experiments is based on the robot actions.

2.1.1 STRIPS

A problem in STRIPS is represented by $P=\{F,O,I,G\}$

- $F$ is a set of atoms. The states $s \in S$ are atoms from $F$. 


Chapter 2. *Automated Planning*

- **O** is a set of operators. These are the actions \( a \in A(s) \).
  - Each operator \( o \in O \) is represented by:
    - Precondition list \( \text{Pre}(o) \in F \).
    - Effects list that can be Add type or Delete type:
      - \( \text{Add}(o) \in F \).
      - \( \text{Delete}(o) \in F \).

- **I** is the initial situation where \( I \in F \). The initial state \( s_0 \) is \( I \).

- **G** is the goal situation where \( s_0 \in F \). The goal state \( s_0 \) is \( G \).
Chapter 3

Logic-Based Finite-State Machines

3.1 LLFSM Description

Finite-state machines are a very useful mechanisms to describe the behavior of a system. They also are comprehensible abstractions for humans about the operation of machines and equipment. Most of us understand that a microwave is either cooking or not, and if one opens the door, the system moves from the state of cooking to the state of not cooking. They are extremely pervasive in describing behavior of components and embedded systems, as well as the behavior of software. They are the base of many commercial tools for modeling behavior (popular commercial tools widely used in the industry include QP\textsuperscript{TM} [Samek, 2008] StateWORKS [Wagner et al., 2006] and a simulator that integrates with MathWorks\textsuperscript{R} StateFlow, like Symlink). They are also part of the UML for modeling software systems (usually describing the behavior of a software component).

Me, in collaboration with MiPal, describe here a model-driven development (MDD) approach to configure and/or compose the behavior of robots, and within this approach, I integrate planning. There are some particular aspects of our logic-label finite state machines that constitute crucial elements. These elements are very distinctive and I believe are essential to facilitate some aspects. In particular, even if an arrangement of several is concurrently executing them, the semantics for their execution corresponds to a single thread; and this aspect enables model checking by standard tools for such formal verification (with tools like [Cimatti et al., 2000]). Thus, I proceed to describe our now.
Chapter 3. Logic-Based Finite-State Machines

The description will follow UML’s because this is a very prevalent model derived from the model [Harel and Naamad, 1996; Harel et al., 1990]. Our model consists of a set $S$ of states, and a transition function $T : S \times E \rightarrow S$. There is a distinguished state $s_0 \in S$, named the initial state. However, the first very important distinction, is that $E$ is not a set of events. Our model is not event based. The set $E$ of labels for transitions is a set of expressions (and in fact, queries), whose evaluation is either true or false. This is why we refer to them as logic-labeled. Transitions may include a reasoning component, and as such, the transition could be the invocation of a reasoning agent, or an expert system, that will determine the veracity of the query.

In many case studies, in MiPal, we have used a common sense non-monotonic logic, Defeasible Logic (DPL)). Because of our participation in RoboCup, we have used models of logic to express what is the off-side rule in soccer [Billington et al., 2007], but our involvement in RoboCup@Home resulted in a case study to describe a common-sense scenario (with non-monotonic logic) for when is dangerous for a senior lady to face a stranger [Billington et al., 2007] and to describe hands of poker [Billington et al., 2009]. For example, logic becomes a very useful tool for communicating what criteria classify a soccer player’s position as on-side (not off-side), or a set of 5 cards as a full-house. This enables declarative knowledge into the behavior. As another example, in the image analysis of robotic soccer, when a blob of orange is a ball [Estivill-Castro and Hexel, 2013] Namely, in very succinctly that a blob of orange is to be considered a ball only if it is reasonably square and there is sufficient density of orange pixels.

We have emphasized this example because the domain knowledge of what to classify as a ball is not a series of sanity checks embedded in imperative language (typically that is cumbersome to debug, maintain and update) and that becomes a black box to the designer of robotic behavior.

When a executes, it issues the queries out of the current state in a deterministic order. That is, in our modeling language, the transitions out of a state are in an ordered list. States have 3 sections, , and . The section contains the actions to be performed upon arrival to the state; while the section executes actions upon departure of a state. The and sections will be executed once and only once in each state. The actions are only executed if the list of transitions out of the state is exhausted without any expression (logic query) resulting in the value true (i.e. no transition fires). In that case, after the internal actions are completed, execution for that returns to re-evaluation the queries from the beginning of the list of transactions. A ringlet is the process of evaluating the list of transitions in sequence and performing the section if a transition fires; or reaching and evaluating the section. If it is the first time execution arrives to the state, the ringlet includes the section.
Figure 3.1: Section of the vision pipeline to recognize a blob of orange as a ball.

```d
% BallConditions.d
name BALLCONDITIONS.
input {badProportionXY},
input {badProportionYX},
input {badDensityVsDensityTolerance}.

BC0: {} => is_it_a_ball.
BC1: badProportionXY => ~is_it_a_ball, BC1 > BC0.
BC2: badProportionYX => ~is_it_a_ball, BC2 > BC0.
BC3: badDensityVsDensityTolerance => ~is_it_a_ball, BC3 > BC0.
output {b is_it_a_ball, "is_it_a_ball"}.
```

Figure 3.2: Theory that defines when a blob is a ball.

Figure 3.1 illustrates these ideas but a formal semantics and more details can be found in other publications [Estivill-Castro and Rosenblueth, 2011; Estivill-Castro and Hexel, 2013] Note that assignment statements and arithmetic (from , , or ) are the actions in the states (Figure 3.1). The transition is_it_a_ball is the query to the engine whose knowledge base is the rules in Figure 3.2.

The executable model is also supported by a model of communication that emphasizes loose-coupling. That is, the variables used are of several types, managed under a whiteboard [Hayes-Roth, 1988] (or also repository [Sommerville, 2010]) architecture. Namely, a state-based variable is a variable whose scope is only the state where it is declared. An internal variable has all the states of an as its scope. An external variable is a variable shared by several . Since an arrangement of is precisely used to have concurrency, the access model of external variables is critical.

In particular, sensors always write their values (at their refresh values) in the corresponding whiteboard variable. This has no conflict, as the particular sensor is the only writer. Actuators always read their parameters from the corresponding variables on the whiteboard. That is actuators are always readers. But any other type of processing
on the robot, and the inter-module communication between the behaviors configured as carry out reads and writes into whiteboard variables. But recall that the arrangement is scheduled deterministically. Each performs a predefined number of ringlets before it passes the execution token to the next in the ringlet. All external variables are read from the whiteboard into local clones at the start of the ringlet, and no further reads take place. Writes are performed immediately into the external variables as copies of the local clones. Therefore, there is no issue of critical sections or additional need for concurrency control.

From the perspective of a software architecture, the whiteboard may seem to some readers as a choke point. However, the sequential scheduling of the arrangements of the guarantees no concurrency issues as we stressed already (there is never an waiting for another). In our Nao robots, the whiteboard architecture has proven to be able to run as fast as the DCM cycle well below the speed of sensors and actuators. That is the local-whiteboard copes perfectly well and it does not constitute a bottleneck. The distributed architecture treats remote-whiteboards as local copies which are updated by the network as if they were sensors, and thus, while the belief on the local copy may be out of date because of network delay, that is no obstacle for efficient coal execution (and well designed robotic software for mission critical response is better based on this time-trigger architectural approach that an even-driven approach).

Form the software engineering point of view, the whiteboard constitutes a publishers-subscribers pattern [Gamma et al., 1995; Larman, 1995] that reduces the number of interfaces for communication. That is, any new module just need to know/use the interface to the whiteboard, and not the interfaces to the other modules.

Finally, one more point regarding the possibility of executing across several CPUs (and several robots) linked over some networking infrastructure. The formal semantics we adopt is a message broadcasting model (no connection; in fact, our prototype implementation uses UDP). Listening to the messages sent across are treated as a sensor. If the message arrives, the corresponding write happens in the local whiteboard. If the message does not arrive, is analogous to a sensor not detecting something (note that every sensor has a usage rate) and not modifying the value it reports.
Chapter 4

Incorporating Planning into the FSMs

4.1 FSM plan control

While this architecture allows very flexible constructs (and in particular it is easy to create behaviors based on feedback-loop control, open-loop control, reactive systems, hybrid-reasonings architectures, and behavior-based control), the challenge is how to flexible incorporate the large body of knowledge from planning. In fact, planners have become standard pieces of software that we would like to smoothly integrate into this semantics of . We actually argue for simplicity here. That is, such integration should not only be standard, but simple to maximize applicability and to ensure adoption.

Because now inputs to a domain-independent planner can be provided in the Planning Domain Definition Language PDDL [McDermott, 2000; Fox and Long, 2003;2006], we integrate different planners as long as they conform to such standardized planning language. We also need to integrate the standardized output so that the resulting fully-ordered or partially-ordered sequence of actions is transmitted to the other elements of the architecture that execute them.

4.1.1 The “plan, execute, re-plan” cycle

The use of a planner in a robot can be organized in a cycle where the robot control provides a planner with a planning request (usually consisting of the robot’s current state, and a goal state). Sensing is involved in checking the success of each action while executing the plan, if the action fails (as perceive by the sensors), the belief the robot
has for its current state is used in a new planning request (re-planning stage) and the robot moves into execution again. Under the name of reactive executive monitoring [Carbone et al., 2008] a reactive planning engine [Beetz and McDermott, 1994] monitors the system’s low-level states against a declarative model of the robot’s functionality, while continuously performing sense-plan-act cycles [Musliner et al., 1993; Williams and Nayak, 1997; Muscettola et al., 2002; Finzi et al., 2004]. In the planning stage, the reactive planner finds a plan (a sequence of robot actions) under some parameters; which may include a planning horizon. Such a parameter balances long planning times with a reactive behavior. During the monitoring, the belief of the robot can be contrasted with operator directives and enable supervised autonomy [Haigh and Veloso, 1998] and integration of intervention by a human operator [Finzi and Orlandini, 2005]. This is known in planning as the “free space assumption” that enables classical planners to work in many complex environments [Bonet and Geffner, 2011].

4.1.2 Planning module API

I now illustrate how our architecture constructs this paradigm. I provide a module that enables several planners to be integrated with the whiteboard architecture through the following API.

\textsc{load\_environment}(a map abstraction) : A map abstraction is provided to the robot. It contains a representation of a real map and the behavior of the dynamic adversaries. We can have as many adversaries as we want, and the behaviors of each one can be completely different.

\textsc{construct\_pddl}(environment) : Once the distribution of the map has been loaded and the environment is constructed taking into account all entities, our program constructs a PDDL that corresponds to that environment and that can be used by the robot.

\textsc{load\_pddl}(planner description) : This enables the planner to retrieve and load a planning problem in PDDL. If there is no file with that name or it does not conform to PDDL, an error is posted to the whiteboard; otherwise, success is reported.

\textsc{load\_planner}(a planner) : This enables to select a planner. In our prototype implementation we currently can chose between a regression planner or the planner [?]. The planning module posts to the whiteboard either success (a known planner was provided) or failure.

\textsc{start\_planner}(depth) : This starts the planner with a certain maximum depth of actions (a horizon). The depth parameter is optional, and if not supplied, plans
of any finite depth are sought. The planner constructs a plan. If a plan exists, confirmation is posted to the whiteboard. Failure is reported when there is no sequence of actions from the source to the goal.

**NEXT_ACTION(rank)**: This request the rank-th action in the current plan. Note that the planner responds with also the action numbered (and the action parameters). This provides some robustness to lost exchanges when we distribute the whiteboard over a network and use UDP. We remark that although the next action may be one from the perspective of the planner, it may actually consist of several action/activities for the robot. For example, an action for the high-level planner may be “approach the goal” which in the robot may trigger a behavior that concurrently tracks the goal, and controls the robot movement in the direction of the goal (or the “approach the goal” may also consist of a sub-plan).

**IS_OBSTACLE_KNOWN(position)**: This reports to the planner an obstacle at the supplied position. This may be an obstacle found along the way or an explanation for why the last action failed. The planner responds whether this obstacle was detailed in the problem description or if it is an obstacle the planner was not aware of. In the later case, the planner updates the problem description to now include the obstacle.

**RE-PLAN(source, depth)**: This planner finds a new plan as with **START_PLANNER**, but from a new source.

Our API provides other tools to convert formats and description of plans or planning problems, but for the purposes of this paper this shall suffice. Also this API allows the planning in an environment where the original map (or problem description) does not have all obstacles (and the robot may discover new ones during execution). New obstacles may trigger re-planning. Fig. 4.1 illustrates the exactly as used in our robots, and in the spirit of Model-Driven Engineering that is used to execute the “plan, execute, re-plan” cycle for our examples with two robotic platforms navigating an environment with some known obstacles but also some unknown obstacles. We omit the few initialization states and transitions for selecting a planner and loading the problem description for simplicity of the figure. We note that the also interacts with the motion module by issuing the motion commands and then receiving from the motion module whether the motion completed or an obstacle was found.

I emphasize that the user develops the behavior graphically and directly as presented in Fig. 4.1.
4.1.3 The inclusion of policies

The first level of interleaving the performance of actions with also planning is the inclusion of policies. Note that while a robot is carrying out an action in the physical world, CPU cycles may become available for other planning. Planning can be complemented by policies; these are default actions to be executed when there is no time or no desire to invest time in re-planning. It may happen that the robot is displaced away from the plan it was carrying out, and as a result, it finds itself in a current position that is not any of the intermediate steps of performing a plan. The robot knows continuing the plan is fruitless, but it may also be wasteful to wait until a new plan is found from the current position. Better to request for a default policy. This is some action the planning module can find rapidly (or it was already calculated as part of the original plan). Policies provide a rapid answer for all those positions of states that are not in the trajectory defined by the plan from the source state to the goal state. Carrying out a policy may be useful while also re-planning after being derailed from a plan.

Note that this new level implies that planning module is executing concurrently with the . For the full implementation of this, we require some calls in the API of our module to deal with concurrency. But for now, to handle policies, our API for our planning module includes the following call.

\texttt{generate\_Policy(heuristic)} : This request the asynchronous generation of policies using the given heuristic. The planner now finds at least one action to be executed.
for each state that is not in a path from the initial state to the goal state.

### 4.1.4 The “execution-driven” cycle

As suggested above, robotic environments suggest to plan-while execute approach [Teichteil-Konigsbuch et al., 2011]. Namely, in the plan-execute cycle, the planning module remains idle while the robot is performing a plan. It only resumes planning once the robot has found an unknown obstacle. This is suitable for static environments where we rarely encounter unknown obstacles. However, in a much more dynamic environment, long plans will rarely be useful, specially the tail of the plan. In such settings, the planner could be concurrently exploring alternative plans (and not just policies) while the execution of the current plan is going on. For this, a model of action failure is required. Such model provides a probability distribution of the results of actions, and thus, the planner could be provided planning request where the source are the most likely outcomes of an action. We note that we do not need the full description of a partially observable Markov decision process (POMDPs).

We can still perform reasonably well with a deterministic planner, we just need to anticipate that some actions may fail more frequently than others.

That is, part of the problem description may be information that describes the likelihood of certain actions succeeding. In other words, for some planning problems, we may have information about the effects of an action and this may be represented by a distribution over a set of possible resulting states from the action. In this case, the may actually issue further planning request as soon as it start executing a given plan. It does so for the second most likely outcome of the first action in the plan, and it may maintain a data structure to represent alternative states that may result from an action.

Also, the planner builds new plans from a source state close to the current state, and places them in a library of plans indexed by source (we take initially a task where the goal is not changing). When the current plan fails, a planning request may be resolved faster by finding a plan in the library that by actually planning. Here, the planning module also manages a queue of planning requests. For this approach, the API of our planning module is the following extension of the previous API.

```plaintext
STOP_PLANNER() : This request removes all planning request from the planner’s queue.

If working on a planning request, this is aborted.

RE_PLAN(source, target, depth, priority) : The planner places this planning request

in its queue with the given priority. The depth and priority are optional; if depth is
```
not supplied, then arbitrary long plans are explored, and if priority is not supplied then the request is placed at the end of the queue.

**START_PLANNER**(depth) : This starts the planner to work on the queue of planning request. The depth parameter is optional, if supplied it overwrites the corresponding parameter in all request in the queue.

**GET_EFFECTS**(rank) : This obtains the set of possible states after the rank-th action labeled with a likelihood value; that is we obtain the probability distribution of the effect of the action.

Also, now the IS_OBSTACLE_KNOWN call may impact on the plans in the library of the planner.
Chapter 5

Model-Checking Behaviors

5.1 Formal verification

Perhaps the most interesting aspect of this approach is the formal verification using standard tools from model-checking. Model-checking consists of automatically verifying properties of a model [Clarke et al., 2001]. In this particular case, we can verify the finite-state machine of Fig. 4.1 meets correctness properties. Since model-checking consists of an exhaustive, almost mathematical proof, it is the highest standard of formal verification. Also, our models (our ) can be simulated, as was previously discussed by using our bridge with .

We will use the common package [Cimatti et al., 2000] which supports the expression of properties in CTL (Computation-Tree Logic [Clarke and Emerson, 1981]) or alternatively in LTL (linear temporal logic or linear-time temporal logic [Huth and Ryan, 2004]).

Some examples of the properties that can be ratified using are the following.

1. If an obstacle is detected, the robot is not moving and the obstacle is unknown, the will certainly request re-planning.

2. If the goal is reached, there is no more issuing of motion commands.

3. After the goal is reached, there will be no more request for re-planning.

4. As long as the goal is not reached, a request for another action to perform will occur infinitely often.

The Kripke structure for Figure 4.1 has 18,473 transitions and close to 2,304 states, but is manageable by today’s laptops. The interpreter of the by the team offers an
Figure 5.1: NUSMV coding of the property that re-planning is certainly requested when an unknown obstacle is detected and the robot has stopped moving.

The formula is uses LTL. The variables of the form $\text{\$\$}$ are the external variables in the whiteboard, while the $\text{\text{pc}}$ variable indicates the position at the execution of a ringlet. In this case there is only one so the $\text{\text{pc}}$ has always a value with prefix $M0$. The formula says that, if an obstacle is detected, the robot is moving and the obstacle is unknown and we are in state 3 past indicating a motion, then either one of these conditions changes or in three transitions on the Kripke structure we will in the section of the re-planning state (state number 4).
Chapter 6

Adversarial Dynamic Environments

6.1 Deterministic Planning in Dynamic Environments

Logic-labeled finite-state machines (LLFMs) are models of behavior that use the ubiquitous model of state machines widely used for representing software behavior [Rumbaugh, et al, 1991], and the behavior of embedded systems (with tools like MathWorks® StateFlow with Symlink). However, as opposed to being event-driven, as are the current version of finite-state machines in UML [Mellor and Balcer, 2002], they use the other model of time suggested by Harel [Harel and Naamad, 1996] where transitions are labeled by statements of a formal logic. For robotic systems, logic-labeled finite-state machines have been used in the language XABSL (where the transitions are labeled by decision trees[M. Lötzsch et al., 2004]) and with Defeasible Logic (DPL) [D. Billington et al., 2011] The version using has also been used to model embedded systems as arrangements of can model concurrent behaviors. Moreover, the arrangement of is scheduled deterministically enabling formal verification with standard model-checking tools of software for embedded systems [Estivill-Castro and Hexel, 2011], [V. Estivill-Castro et al., 2012]

Our argument here is that, such logic-labeled machines can describe the behavior of other components of an environment for a robot. Moreover, while deterministic planners may have been conceived for environments where the solution is a sequence of actions structured as a plan for the components of the environment that there is control, I suggest that we can use such deterministic planners even for environments where there is dynamic behavior of environment elements for which the planner and the robot do not
have control — as long as those other components have their behavior described by logic-labeled finite-state machines. That is, I will use deterministic planners in deterministic environments, but the environments I consider are much more dynamic as they will include other objects that act on the environment. The idea is similar to the planning happening in Cooperative $A^*$, (that moves from reasoning on space to reasoning on time and space), but rather than having a dedicated planner with $(position, time)$ states, I create the description in PDDL and any PDDL-planner can attempt\footnote{As I will discuss later, while PDDL-description facilitates using any PDDL-planner, because of their search strategy and driving heuristic, some planners will be more effective than others.} to build a plan.

To illustrate this point, I again refer to the Sokoban grid-based environment. This is considered an extremely challenging environment for deterministic planners. At the international planning competition, some instances have been recycled to the next competition as all solvers performed poorly in the puzzle. However, only one robot moves around and solvers consider only the moves of this robot. As a result, only the effects of the plan modify in any way the environment. I illustrate this suggestion of a dynamic environment by considering other adversaries in the environment that also move. An example of such adversary could be another robot in the environment for which our controllable robot only knows its deterministic behavior as it is expressed in a logic-labeled finite-state machine. This is an extension on Sokoban that makes the environment dynamic.

Robots navigation behavior as a is in fact natural for patrolling robots and can be synthesized out of specifications from a temporal logic. In fact, our robot could actually know the LTL specification of the other robots and use such approach to infer the of the other adversaries in the environment. For illustration purposes, Fig. 6.1 presents an example of a for this extended grid-based environment. This agent loops around 4 states, successively moving one cell West, one cell South, one cell East and one cell North in the grid environment. Clearly, grid-environments are an abstraction (for planning and reasoning) of the real-world environments where robots navigate in that lower level behaviors to achieve grid-environment abstractions such as moving a robot from one cell to another, or from one room to another).

Transitions are labeled by logical expressions, in this case they are simple. However, this logic-labeled finite-state machine is sufficient for illustrating the process of describing the dynamic environment entirely in PDDL, in order to apply deterministic planning software packages. For this example, I let the granularity of the adversary’s time-step to be the same as for the mobile robot under the controller. However, if the time steps of the adversaries and robot are different, we can just choose a common divisor of the time steps for the PDDL description to reason about the states of the world.
Figure 6.1: A that encodes the behavior of a simple adversary that loops in 4 positions.

Figure 6.2: File that provides the PDDL domain description for the adversary

For each motion on the grid by an adversary the usual actions that describe the motion of the robot must be re-classified into 4 new action descriptions. These correspond to the following 4 situations.

1. move-is-possible. This is the first case, when the robot and the adversary do not collide. This collision is not existent because the context, verified in the pre-conditions, is as follows.
(a) The robot does not perform a move to a grid-cell currently occupied by the adversary.

(b) The adversary does not perform a move to a grid-cell currently occupied by the robot.

(c) Both, robot and adversary, do not perform a move to the grid-cell occupied by the other.

The PDDL code for this is illustrated in Fig. 6.2 under the first action labeled move-is-possible. In general, I use a predicate connect\((p_0, p_1)\) between two positions to indicate that the robot (or the adversaries) can move from position \(p_0\) to position \(p_1\). The collection of connected positions is the static description of the environment, in other words, the map of the world describing open space and fixed obstacles. The predicate next\((p_0,s_1,p_1,s_2)\) encodes all the position and state changes of the logic-labeled finite-state machine by the adversary. For example that the adversary changes state as it moves East is expressed by declaration of the PDDL problem of the following form.

\[
\text{next } \text{pos}_0 \text{ pos}_1 \text{ c0 } \text{pos}_1 \text{ c1}
\]

The current position of the robot is the position \(p_0\) for which the predicate at-robot\((p_0)\) is true. The current state \(s\) and position \(p_0\) of the adversary is the pair \(p_0, s\) for which at-adversary\((p_0, s)\) is true. Then, the parameters of the action in the case move-is-possible are the current position and the next position of a robot’s move while for the adversary are the current (position,state)-pair and the next (position,state)-pair. As a precondition for the action I do need to check these positions are current and that the map allows to move from the current to the next position for both the robot and the adversary (the predicate connect does this for the robot while next\((p_0,s_1,p_1,s_2)\) implies connect\((p_0,p_1)\), and handles the free space of the robot). I also need as part of the precondition the 3 cases of this situation. For example, the following test in the precondition

\[
\{ (\text{not}(=\text{pos1 } \text{pos4})) \}
\]

checks that the robot and the adversary do not share the same grid-cell as the target of their respective moves.

If the precondition holds, the effects are that the robot is in its new position and the adversary is as well in its new position and state (and their are not in their earlier positions). While there is an effect on the cost, we delay the justification until we complete the four cases.
2. move-is-not-possible-adversary. This is the same action for our robot, but in this case the adversary’s next position in the grid-cell map is blocked by the robot. Thus, the adversary must wait in its current position. That the robot blocks the adversary move is now the predicate (=?pos1 ?pos4) of the action’s precondition in the PDDL of Fig. 6.2.

3. wait-and-adversary-moves. Again the same action of the robot is qualified by the condition that the robot cannot move to a cell occupied currently by the adversary. However, for planning purposes, we do have the effect that the adversary will carry on its behavior as per its logic-labeled finite-state machine, and therefore, it may vacate the area where the robot desires to move. Nevertheless, the robot would have to carry a wait action, at least in this time step.

4. wait-and-adversary-cannot-move. In this case, not only the action of the robot is blocked by the adversary, but the adversary’s behavior is also blocked by the robot. In this case, the effect is that both remain in the same position.

I hope the reader appreciates the generality of the derivation of the PDDL domain description from the behavior of one or more adversaries as long as they are specified by logic-labeled finite-state machines. Also, the PDDL is generic in the structure of the map by the connect predicate. If moves/actions of the robot are more complex on what, when and where is an action applicable, the corresponding predicates for these specification would be replacing (expanding) this predicate.

The last point to discuss is the fact that we have associated weights with the four qualifications placed at each action of the robot. The actions when the robot actually moves are assigned a slightly larger cost than those where the robot waits. We can anticipate that normally not performing an action and remaining idle is less costly (energy consumption, for example), than actually performing an action (a move that may require energy for the motors). However, this is also a condition to assist the solver in focusing on the types of plans we would expect to be realistic. If no cost is provided, PDDL solvers produce plans that may be regarded as anomalous, because they may contain spurious sequences of actions; for example, they move out and in from one position rather than wait for two time steps in the position.

6.1.1 Illustration

To clarify the earlier presentation I use a the running example of Fig. 6.1 and its PDDL (see Fig. 6.2) with two scenarios where the initial position of one patrolling adversary is
known. I thus suggest an environment as per Figure 6.3\(^2\). This map has a fixed obstacle on (0,2) and the goal is always at (0,4). In the first setting, the initial position of the automata (the adversary) is (0,3) and its patrolling behavior is as per Fig. 6.1. We place our robot in (0,0), and configure different problem instances by selecting different cells as the initial position of the adversary. For illustration, I also specify in the domain instance that the only actions for the robot are rectilinear motions of one grid cell in the map.

Now, using our coded domain of Fig. 6.2, and our planning API for a robot, then solvers like regression or lama [Richter and Westphal, 2010] find a plan that goes along the following sequence of actions while the automata carries out the corresponding moves.

<table>
<thead>
<tr>
<th>robot</th>
<th>adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(1,4)</td>
</tr>
<tr>
<td>(1,4)</td>
<td>(0,4)</td>
</tr>
</tbody>
</table>

However, if the adversary’s initial position is (1,3), then the plan found by the solvers regression and lama has the following form (where the adversary is forced to wait).

<table>
<thead>
<tr>
<th>robot</th>
<th>adversary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(1,4)</td>
</tr>
<tr>
<td>(1,4)</td>
<td>(0,4)</td>
</tr>
</tbody>
</table>

But if the adversary’s initial position is (1,4), then our solvers’ plan must include an action to wait.

\(^2\)The original image of the e-Puck was released by its author Stéphane Magnenat on http://commons.wikimedia.org.
6.2 Planning API for Dynamic Environments

Our first step is to emulate the behavior of the adversaries (represented as a logic-labelled finite-state machine) into the spatial map abstraction of the world. The constructed sequence of maps contains the elements and distribution on the environment and this also constitutes an abstract representation of the behavior of the adversaries. In a sense we get each spatial map also for each time-step in a temporal dimension. Once we obtain the map abstraction and the adversarial behavior, the PDDL is automatically constructed and we can invoke the solvers to generate plans for the environment.

This API allows integration of the planning on board of a robot for environments that can differ from the initial knowledge (the original map or problem description). Unknown fixed obstacles are discovered by the robot and re-planning adapts plans to new situations. Our API provides the tools to perform continuous sense-plan-act cycles [Musliner et al., 1993; Williams and Nayak, 1997; Muscettola et al., 2002; Finzi et al., 2004] as well as a plan-while execute approach. In that sense, we do not depart from the common “free-space assumption” where we can plan until we must revise because we discover an obstacle or an adversary.

We underline here with another example, that even with the suitable sensors model and localization algorithms, a robot that can plan and act reliably those actions, faces challenges with deterministic (re-planning) planners when the environment presents dynamic elements if it does not apply the translation we propose. A dynamic element is one that does not sustain its position all the time. It can move through the environment following a predefined behavior. An example where the plan, execute, re-plan cycle fails is provided in the example problem represented in Fig. 6.4. Here, the adversary moves from cells (2,2) to (2,1), from (2,1) to (2,2) and so on, one cell at each time. The robot is on (0,1), it does a forward movement for the goal to (1,1), here at (1,1) the sensors detect an obstacle blocking (2,1). A message asking if that obstacle is known is posted into the whiteboard and two things can happen:

1. The planner module has a library of plans where the current robot position corresponds to a position in the path of one of that plans. So now, the planner module
Figure 6.4: An environment where deterministic planning with obstacle sensing and re-planning fails.

... gives another possible plan to the robot that can reach the goal following another path.

2. The planner module does not have a library of plans, or another available plans does not solve the problem. So the planner module re-plans again in order to find a new path to the goal, taking into account the new discovered obstacle.

In both situations, the robot will move to the next position (1,2) and it will detect the cell (2,2) as blocked, — the same as the other case, but now the adversary has moved to (2,2). The same problem will occur, the planner will not find a plan unless it models the behavior of the moving adversary and performs a wait.

Thus, enabling dynamic elements with deterministic planners requires the construction we have described earlier.
Chapter 7

Experiments and Results

7.1 Experimental evaluation

The success of this API design and the software architecture has been demonstrated by using several different domains where robots must navigate an environment with known and unknown obstacles. The overall behavior is encapsulated by the deployment of an that invoke the necessary planner. Such behaviors are also independent of the particular robotic platform. That is, the responses from the planning module are significantly high level. As a result, we can perform planning with the same architecture and the same behavior on different robotic platforms. For illustration, we have videos of an 1 robot with tethered control (in the form of a differential robot), a 2 humanoid robot and also the e-puck robot (differential robot) inside the Webots simulator. The behavior that invokes the planner is the same with the option of choosing the planner it we suspect more alternative paths are required besides just one. The execution of different plans with the regression planner is illustrated on the and on the robot in a video used for classification for RoboCup-2013. (see vimeo.com/mipalgu/qualification2013 or yououtu.be/cQgCrqRznCo from 1min-30s to 3min). The execution of plans where the planner is underneath is demonstrated in the videos for the 3 simulator (refer to video showing alternative paths yououtu.be/-mvppFPWfMU). The plan, execute, re-plan cycle is driven by a like as per Fig. 4.1. The video is accelerated after a few steps and a speech module is used to record the actions received for each action request. The dimension of each square is equivalent to 40cm.

1LEGO MINDSTORMS NXT, is a configurable robotic system distributed by LEGO (mindstorms.lego.com).
2NAO is a programmable, 58cm tall humanoid robot distributed by Aldebaran Robotics (www.aldebaran-robotics.com).
3is a development environment used to model, program and simulate mobile robots distributed by Cyberbotics (www.cyberbotics.com).
The execution of plans where the lama planner is underneath is demonstrated in the videos for the \textsuperscript{4} simulator. The video is accelerated and a speech module is used to record the actions received for each action request. The length of the side of all squares is equivalent to 40cm. For evidence of the proposal here we actually provide the video \url{http://www.youtube.com/watch?v=6MJ1I0ZTeVc} where we display 3 scenarios. All were actually computed by the lama planner as the \textit{regression} planner actually fails to find a plan. This means that the state explosion of the environment here surpasses the heuristic of this solver. This aspect illustrates the relevance of our API enabling the selection of a different PDDL-solver. In the first scenario we keep one adversary fixed, while another does move under a deterministic behavior (specified by a ). The for this adversary is significantly more complicated that the one in Fig. \ref{fig:6.1} (the adversaries repeat position but moving in different directions). The second part of the video shows the scaling to even more adversaries, as both become moving adversaries, and we have chosen one of the lama plans that illustrate the choosing of all actions possible by the robot. Finally, we show that under static planning conditions, all classical planner would guide the robot to a crash. However, the planning control has been factored out to a common with the option of choosing the lama planner if we suspect more alternative paths are required besides just one.

The overall behavior is encapsulated by the deployment of an that invoke the necessary planner. That is, the responses from the planning module are significantly high level. As a result, we can perform planning with the same architecture and the same behavior on different robotic platforms. For illustration, we have videos of an \textsuperscript{5} robot with tethered control (in the form of a differential robot), a \textsuperscript{6} humanoid robot and also the e-puck robot (differential robot) inside the Webots simulator. From 1min-30s until 3min, our video used for classification for RoboCup-2013 illustrates the execution of plans with the regression planner on two architectures; namely on the and on the robot (see \url{http://vimeo.com/mipalgu/qualification2013} or \url{http://youtu.be/cQgCrqRznCo} from 1min-30s to 3min). Another video (\url{http://youtu.be/-mvppFPWfMU}) shows alternative paths for with lama.

\footnotesize{\textsuperscript{4} is a development environment used to model, program and simulate mobile robots distributed by Cyberbotics (\url{http://www.cyberbotics.com}).
\textsuperscript{5} LEGO MINDSTORMS NXT, is a configurable robotic system distributed by LEGO (\url{http://mindstorms.lego.com}).
\textsuperscript{6} NAO is a programmable, 58cm tall humanoid robot distributed by Aldebaran Robotics.}
Chapter 8

Related Work

8.1 Contrast

A similar framework for developing behaviors using have been proposed in the literature. A notable example is [Beetz et al., 2010]. This is a set of tools focused around manipulation and enables the design, implementation, and the deployment of activities under a cognitive model. Although plans are enabled, the integration is not a closely coupled as us, as there needs to be close interaction with a knowledge services engine (KNOWROB) and a reasoning layer (COGITO). Also, is not as generic as our proposal, as it requires the use of its own planning language.

Another interesting alternative is \(^1\). This architecture intends to develop far more than an integration with planning. As such, is based on a working memory with the structure of frames (the role played by our whiteboard), but the entire development is using logic programming. It is developed in and does not offer a simple interface to PDDL descriptions. By its own admission, it is focused on agents systems performing cognitive tasks, one of which may be planning, but not necessarily.

As gains popularity in the robotics community, some of the packages associated with it may do so as well. This may be the case for \(^2\). However, is just a series of templates, it does not have tools support and does not offer the clear sequential semantics of the logic-based discussed here. It does not offer any specific tools for including planning as we have enabled here with the API described in the earlier section.

\(^1\) is a general cognitive architecture for developing agent oriented systems exhibiting intelligent behavior distributed by the University of Michigan (sitemaker.umich.edu/soar.)

\(^2\) aims at enabling rapid prototyping of complex robot behavior www.ros.org/wiki/smach.)
Similar to the Boost FSM Library\(^3\). It does not enable the transitions to be reasoning queries, and the interpreter does not enforce the sequential schedule of several making very hard to perform model-checking.

Another interesting example is the PLEXIL language [Verma et al., 2005], used in spacecraft missions. It provides and offers a representation for executable plans and it is designed for command executions. It allows plan and re-planning that can be generated On-board or Off-board and the knowledge representation is included in the domain description that includes the actions, tasks, constraints, etc.

On the same way, the Procedural Reasoning System (PRS) [Ingrand et al., 1996], is a set of generic tools and methods used to represent plans and its execution, performing actions and tasks in dynamic environments done by autonomous robots. It allows the use of planners that provide plans to PRS. It is composed by a task graph that represents the executing tasks, a database, that contains the representation of the world and also it is constantly updating, and, as our proposal, a library of plans.

Finally, we must acknowledge that the idea of transitions labeled by queries (and not by events) can probably be traced to the modelling language [Lotzsch et al., 2004; Risler and von Stryk, 2008] for robots and agents where transitions are labeled by decision trees. The there aim at being close to UML’s state-charts similarly as done here. However, there are many issues with UML’s state-charts [Estivill-Castro and Hexel, 2013] that are eliminated with the sequential scheduling used here.

\(^3\) Boost FSM Library offers classes and templates to produce code structures or organized as a (boost-extension.redshoelace.com/docs/boost/fsm/doc/state_machine.html).
Chapter 9

Conclusions and Future Work

9.1 Conclusions

Therefore, we believe we offer a unique API and a unique architecture to integrate high-level planners into robotics systems as an intelligent capability module. Also with a full interaction with other functionalities present in mobile robotics.

This architecture offers a monitoring and previous construction of the world following the PDDL standards and it adapts the plan with its complex tasks to the eventual variation of the environment. Actions are requested by the robot depending on the state and the posterior effects in addition with concurrent actions. Also, the inclusion of policies allows the robot to request for new generated actions depending on the given heuristic and can use plans stored as a library of plans if they are available.

Moreover, the behaviors are constructed using logic labeled finite state machines and we use formal verification of the properties in our FSMS, that it is our model.

The advantages of our contribution are transparency, clear semantics of the schedule of execution, conformity with the PDDL standards and facility to create behavior paradigms where the planning cycle could be overlapped with the acting phases of the robot. We argue for the standarization using high-level planners, so PDDL is very useful for represent and abstract the environments. In addition, our approach has been tested in several robotic platforms doing the same and different experiments with each one. These platforms are at the moment, a Nao Robot, a NXT Lego Mindstorm and the simulation platform Webots.

This paper explores to what the extent a deterministic planner can be used to guide the motions of a robot in a dynamic environment. We have shown that objects that move
in the environment with a deterministic behavior can be integrated into the PDDL descriptions for these types of planners. The general method consists of specifying the states and the transitions of each of such object as connections in the problem specification part of the PDDL. Then, the actions of our robot (as specified in the domain part of the PDDL) are decorated with the preconditions and effects of each of the moving objects.

This general methodology is not without its challenges. However, it already proves to be superior to the approaches that use planner and re-planning by deterministic planners in a dynamic environment. Recall the early discussion regarding the environment of Fig. 6.4

### 9.2 Future Work

After that, I can think on several aspects or ideas to take into account on the future. For both, improve and extend the developed work.

It will be interesting to study the capacity to detect and recognize predefined behaviors done by other moving agents. Using for that purpose observations of the environment done by robot sensors and calculating the probability that a certain adversarial behavior corresponds to a known behavior as in Fig. 9.1. So, the agent can adapt its actions in order to avoid the adversarial patrolling behavior.

---

**Figure 9.1:** The agent can calculate with certain probability the adversarial behavior.
Chapter 10

Bibliography


Barceloan, Spain, 19-21 February 2013. SCITEPRESS Science and Technology Publications.


