A Modular ROS package for Linear Temporal Logic based Motion Planning

by

Parth Pandya

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2013 by the
Graduate Supervisory Committee:

Georgios Fainekos, Chair
Yann-Hang Lee
Partha Dasgupta

ARIZONA STATE UNIVERSITY

May 2013

ABSTRACT

Objective of this thesis project is to build a prototype using Linear Temporal Logic specifications for generating a 2D motion plan commanding an iRobot to fulfill the specifications. This thesis project was created for Cyber Physical Systems Lab in Arizona State University. The end product of this thesis is creation of a software solution which can be used in the academia and industry for research in cyber physical systems related applications.

The major features of the project are: creating a modular system for motion planning, use of Robot Operating System (ROS), use of triangulation for environment decomposition and using stargazer sensor for localization. The project is built on an open source software called ROS which provides an environment where it is very easy to integrate different modules be it software or hardware on a Linux based platform. Use of ROS implies the project or its modules can be adapted quickly for different applications as the need arises.

The final software package created and tested takes a data file as its input which contains the LTL specifications, a symbols list used in the LTL and finally the environment polygon data containing real world coordinates for all polygons and also information on neighbors and parents of each polygon. The software package successfully ran the experiment of coverage, reachability with avoidance and sequencing.

To My Wife & Parents

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Georgios Fainekos for giving me an opportunity to work on this research topic and providing valuable guidance, inspiration and support. I would like also thank Dr Yann-Hang Lee and Dr. Partha Dasgupta for their ideas and feedback as being part of my thesis committee. This thesis was partially funded by the grant NSF CNS 1116136.

I would also like to thank my friends and colleagues at Arizona State University and beyond. In particular I would like to thank Anand Sudhan, Adel Dokhanchi, Bardh Hoxha, Hengyi Yang, Kangjin Kim, Ramtin Kermani, Sangeetha Nandan, Shih-Kai Su, Shashank Srinivas and Tosha Shah for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER1

INTRODUCTION

1.1 Motivation of the Thesis

There is an increase in the pervasiveness of technology in our daily lives. Technology is slowly taking over menial tasks and freeing us for tasks which require higher cognitive abilities. The mechanization of manufacturing industry or our web searches becoming more intelligent are examples of technology easing our lives. One of the major driving reason behind all this technology is software capabilities being brought into the picture. Yet there exists a gap between specifying the work we need to get done and actually extracting the work from the machine.

This problem is attacked in two different ways, first is to have a statistical analysis on the data being provided by the user and provide a reaction accordingly, the second one is being able to get a more clear request from the user to begin with and get the work done. The first problem can be applied to many situations, but a major shortcoming of this method is for each user to provide with some initial data before being able to actually get useful results from the system. The second method has the challenge of designing a good interface to be used for human to machine interactions and prepare software which is robust to ambiguities. This means building the safety critical aspects into the software itself and not have contradictory or unpredictable states in the output.

One of our basic human interaction mechanisms are natural languages. The problem with using natural languages is the inherent ambiguity that lies in them. Linear Temporal Logic is a language that can be used to create a bridge for this gap and trying to produce a more exact mapping between the intention of a statement and reducing the associated ambiguities with its meaning (Clarke, Grumberg, & Peled, 1999). Linear Temporal logic enables us to reason for time over state machines and provides a powerful tool to get closer to deriving concrete specifications from natural language constructs. LTL is used for model checking which is aimed at finding an acceptance state and its reachability for all time. Whether there is a possibility of the system breaking down after "x" number of iterations can be found out because LTL has the ability to describe infinitely often behavior which is otherwise not easily verifiable.

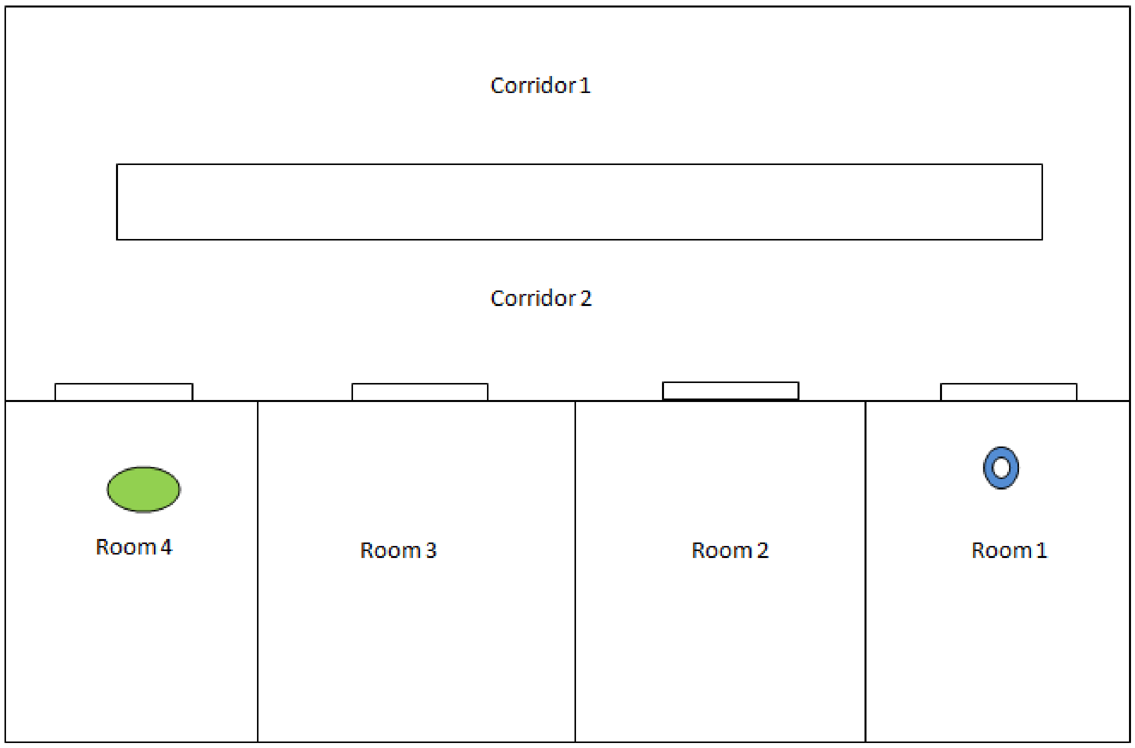


Figure 1: Shows the environment with 4 rooms and 2 corridors.

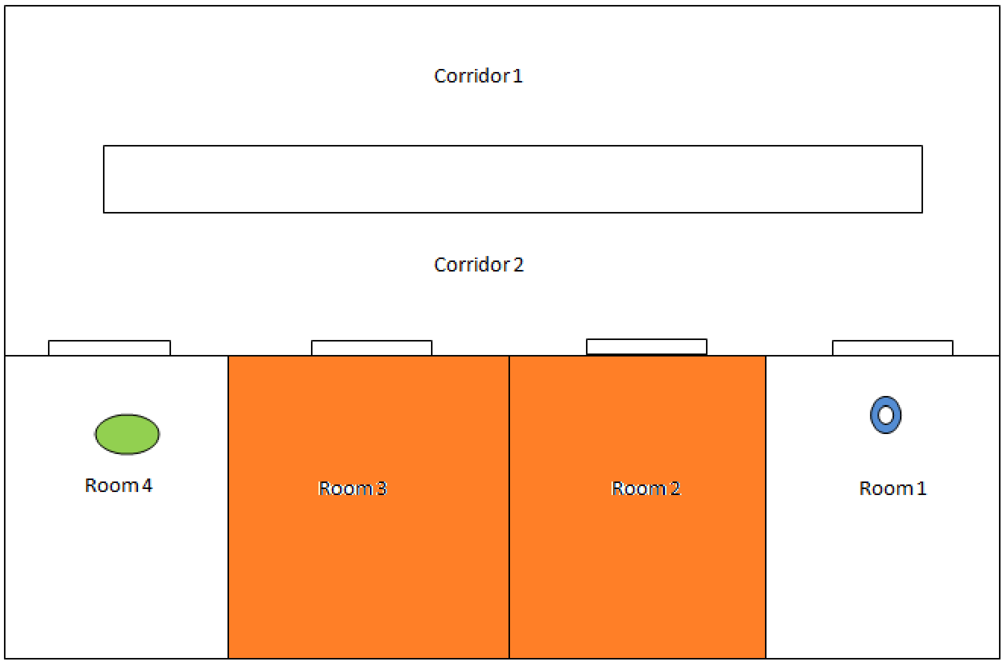


Figure 2: The robot in Room 1 is directed to pick object in Room 4 while avoiding Room 2 & 3.

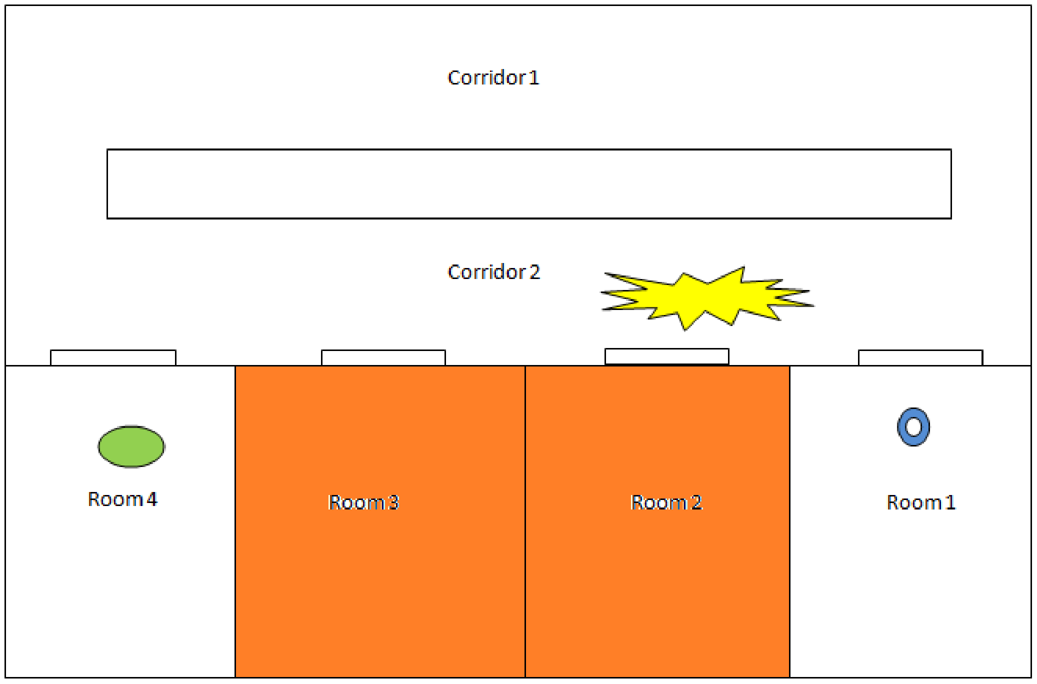


Figure 3: The robot is directed to avoid Corridor 2, Room 2 & 3 while picking the object from Room 4.

The model based specifications in LTL can provide guarantees on whether reaching the final goal is feasible. Consider the following example where we see 4 rooms adjacent to each other connected by 2 corridors in a large environment as shown in Figure 1, a robot is directed to pick an object from Room 4. Now consider a situation where a robot has been asked to perform the previously mentioned task while avoiding Room 2 and 3 as shown in Figure 2. Consider a further situation where some kind of blockage exists in Corridor 2 as shown in Figure 3. We can easily keep incorporating all these new behavior in the specification and ask the robot to complete its tasks using LTL specifications, because to produce these varied behaviors we only have to modify the LTL specifications to guarantee expected results. An availability of natural language to LTL converters (Dzifcak, Scheutz, Baral, & Schermerhorn, 2009) makes it's even easier for a user to provide behavior specific commands to a machine which hold over a period of events. As we can see here it is very easy to incorporate more complex behavior in the system without really having to make extensive software changes on the underlying layers. This leads to flexibility of how the robot is supposed to perform an operation and opens up new venues to interact with machines without having complex understanding of underlying software architecture. Hence producing a motion plan which can guarantee complex event based behavior becomes easier using LTL specifications.

1.2 Contribution of the Thesis

The main contribution of this thesis is the creation of a modular software providing an implementation to generate motion plans for an iRobot from LTL specifications in a defined environment with a ROS framework. A ROS based LTL planning system has yet not been devised and this implementation will address this need of creating a package which will be made open source at a later point of time. The software is implemented in C++ using the facilities provided by the ROS framework to communicate to iRobot and stargazer hardware components. The software provides a test bed for researchers and engineers to verify LTL planning algorithms on a physical robot rather than a simulation environment. The modular nature of the software also enables certain parts to be utilized with few changes to it. It is feasible to just get cross-product as an output from the system into a file. The software does many tasks which can be tweaked and enable a variety of different functions. The package will be publically released on the conclusions of a stable version to the ever growing ROS community.

The use of ROS gives it access to a variety of hardware because of availability of open source packages make it easy to adapt code. This implies that a number of sensors could be used for localization along with a different set of robot for motion. The software also contains a use of triangulation in environment decomposition instead of square grids and, use of stargazer sensors for localization to derive accurate location. These features help in generating a faster and more non-intrusive path for the robot to use for its navigation (Kuan, Zamiska, & Brooks, 1985) (Vik, 2013).

1.3 Thesis Structure

This thesis is intended to be a guide for the software developed. It talks in depth on the details that were used to generate robot motion and its controllers from LTL.

- Chapter 1 gives an introduction and a general outline for the motivation behind this thesis project.

- Chapter 2 discusses related literature and the background of the entire thesis project. The direction of doing the thesis project and the reasoning behind taking this particular design route.
- Chapter 3 describes the design of the ROS package that was created and gives details related to the software and hardware architecture that were used. It describes the general outline of the code and its construction along with different modules that were integrated together to implement the code.
- Chapter 4 is related to the interface files that are required as inputs, fields and the reasoning behind them.
- Chapter 5 discusses the results that were produced and the end deliverables created for the thesis project.
- Chapter 6 covers possible future work that can be done based on the package that was created for this purpose.
- Appendix contains the instructions for initializing this software and the related setup required. It also talks about the stargazer map used to localize the iRobot for its navigational feedback from the system.

CHAPTER 2

RELATED LITERATURE

This thesis is going to present the idea of combining LTL with movement of iRobot. The main goal of this project is to generate a controller for a robot by taking Linear Temporal Logic specifications and using them to create a navigation path plan for a robot. In the future we will see an increase in the way technology integrates with our lives. We are seeing a definite increase in robots being around us. As we gain more proximity and use of robots we need to work on being able to communicate with the machines in an effective way. There exists a gap between being able to give clear instructions to a robot from a human perspective. This is because natural languages can implicitly have ambiguity built into them. This is where we need to use LTL to define the specifications which provide a closer alternative to the natural languages.

2.1 Related research

The previous related work is based on a variety of related topics. This project combines different software tools and constructs an integrated package. This section describes the theory behind all the different software packages that were used. The topics are subdivided into the following major points:

- Motion Planning
- Temporal logic for motion planning
- Environment decomposition

2.1.1 Motion planning

Motion planning has many interpretations based on the context from which it is drawn (LaValle, 2006). Motion planning could mean trying to find path to solve a complex puzzle in the perspective of AI or trying to find the shortest path between two cities while using a GPS, or how a robot arm should be manipulated to successfully run its task in an assembly line. This thesis concentrates on the concept of motion planning that is related to motion of a robot from one location to another given an environment with set of events or constraints. These constraints can be based either on the robot control or the ordering of events and their temporal relationship with each other.

There are two approaches to robot motion planning that are considered. The first one is to generate a path for very complex environments with the assumption that a robot does not have control constraints. The other approach is to use complex models for the robot but have very simplistic environments for them to navigate in. Both these approaches have their drawbacks in implementing them as concepts to the real world (Belta, Isler, & Pappas, 2005). The work described in (Belta, Isler, & Pappas, 2005) talks about integrating high level and low level path planning by taking a two pronged approach. First is generating a path plan into discretized set of events which produce a plan for the robot to move in a triangulated environment, second point takes into consideration the low level kinematics of a robot such that they are able to ensure its ability to stay in triangulated spaces as requested by the high level plans. The low level dynamics will ensure that irrespective of the initial condition in a decomposed environment cell, it will be able to maintain the sanctity of staying within the environment cell. This provides an abstraction in trying to generate the controllers needed for a robot to navigate a planar environment consisting of convex polygons.

2.1.2 Temporal logic for motion planning

"Temporal logic is a branch of modal logic." (Kroger & Merz) Temporal logic can be defined as consisting of propositions with standard Boolean operators and some temporal operators (Kress-Gazit, Fainekos, & Pappas, 2009). (Pnueli, 1977) proposed that a program is supposed to be composed of two components which could be used to describe its correctness. These components are called

1. Invariance: a condition has to stay true during the entire run of the program, which means regardless of the current state of program, the program has to generate conditions and ensure that invariant specifications remain true at all times.
2. Eventuality : means a condition will become true eventually which means at some point of time in the execution trace in the future the condition holds.

There are also various types of temporal logic that have been explored (Lamport, 1980). They are based on how time is treated i.e. either time can be treated as all future possibilities happening simultaneously or time being a linear set of events with only one real future. The major difference that comes about from these interpretations is explained as follows, in case of branching time into all possibilities implies every possible future is real and has to be taken into consideration. This interpretation does not apply to linear time logic problems where just one actual future that takes place. This difference is highlighted when we see a statement with "not never " and "eventually" where because of branches we find that "not never not" implies within the system there are some futures where "not never not" statement will be realized but not necessarily for every future which is not the same as "eventually" which implies it being true in all the possible future at some point in time. It was proposed that linear time theory is better suited for applications or programs with requirements of multiple processes running together.

There are many types of temporal logic which can be used such as computation tree logic, LTL and real time logics, etc. the thesis project uses Linear temporal logic. The operators used by LTL are described in Table 1 below.

Table 1: Describes Linear Temporal Logic Operators used.

Operator	**Symbol**	**Operator Type**
Conjunction	∧	Boolean
Disjunction	∨	Boolean
Negation	¬	Boolean
Eventually	◊	Temporal operator
Always	□	Temporal operator
Until	U	Temporal operator
Next	o	Temporal operator

Let us define a region as π_i where i = 1,2,3...,n labeling each individual region present inside the defined environment. A complex set of behavior can be expressed using these operators for the robot navigation such as

- Reachability : the formula $\Box(\neg\pi_1)\wedge(\Diamond\pi_2 \wedge \Diamond\pi_3)$, means visiting locations π_2 and π_3 while avoiding unsafe area π_1 at all times.
- Sequencing : $\Diamond(\pi_1 \wedge \Diamond(\pi_2 \wedge \Diamond\pi_3))$ means a sequence of visits to π_1 first followed by π_2 and then π_3.
- Coverage : $(\Diamond\pi_1 \wedge \Diamond\pi_2 \wedge \Diamond\pi_3 \wedge ..... \wedge \Diamond\pi_n)$ means that eventually reach π_1 and eventually reach π_2 and eventually π_3 and so on.

In (Clarke, Grumberg, & Peled, 1999) the argument behind trying to use temporal specifications for verifying a software system is discussed. It is contended that as software systems become complicated they need to be verified before being released to the outer world. This kind of verification can be done by the use of model checking techniques where Linear Temporal Logic(LTL) can verify the specifications correctness.

In (Fainekos, Kress-Gazit, & Pappas, 2005) it is explained how a closed loop or an open loop controller can be derived from a set of specifications and the environment decomposition models. It is described on how to build an entire path planning system for a robot. As explained in (Fainekos, Kress-Gazit, & Pappas, 2005) we can implement and create a controller for a robot using Linear Temporal Logic (LTL) specifications which define the constraints on the robot in its actuation or its environment. Furthermore, the use of closed loop systems to determine the initial condition such as current location coordinates guarantee the successful run of operations and is a necessary component (Fainekos, Kress-Gazit, & Pappas, 2005, pp. 4885-4890).

2.1.3 Environment decomposition

Motion planning has to inherently deal with representations of the state space. In the physical world we see the state space in a continuous manner but a discrete set of states are used to actually perform the motion plan. This is because continuous space means there is an infinite state space. Hence to counter this problem we convert the continuous space into a discrete set which preserves the original problem (LaValle, 2006). Typically the environment can be represented by polygons. There are a wide variety of polygons such as convex, spiral, star-shaped as shown in the Figure 4: Figure showing different types of polygons.. There are many applications for polygon decomposition such as creating VLSI designs for layouts for lithography (Nahar & Sahni, 1988), pattern recognition of complex patterns broken down into recognizable simpler patterns. It is also used for computational geometry, database systems, image processing and computer graphics to draw the images on screen.

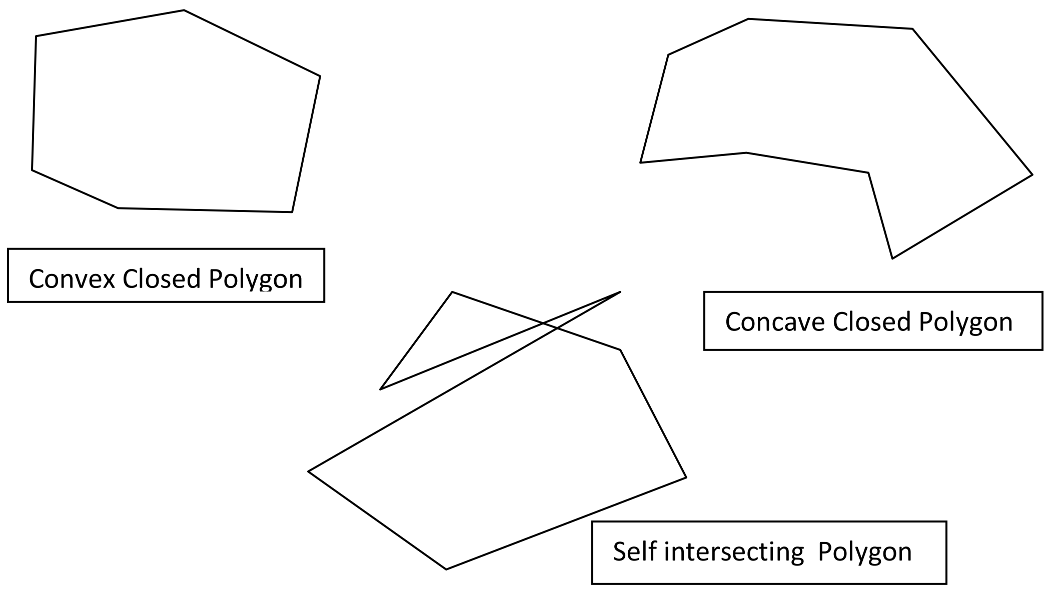


Figure 4: Figure showing different types of polygons.

We are interested in polygons which are called simple polygons with holes that can easily be used to define a large set of environments and then be decomposed to generate simpler primitives. A simple polygon is defined as a polygon which has a measureable area, exactly two edges meet at each vertex of the polygon and edges meet only at their vertices. The polygons are decomposed into cells which are treated as individual states. At a more abstract level the decomposition created are called partitions when the sub-polygons do not overlap except for the boundaries, otherwise they are called covers. We are interested in partitions only because of the need to create discrete unique states from the environment where going from one state to another will cause a transition towards the goal of the motion plan. The decomposition is done by generating smaller primitives within polygon boundaries. The cell decomposition is a problem that has received a lot of attention due to it being computationally intensive.

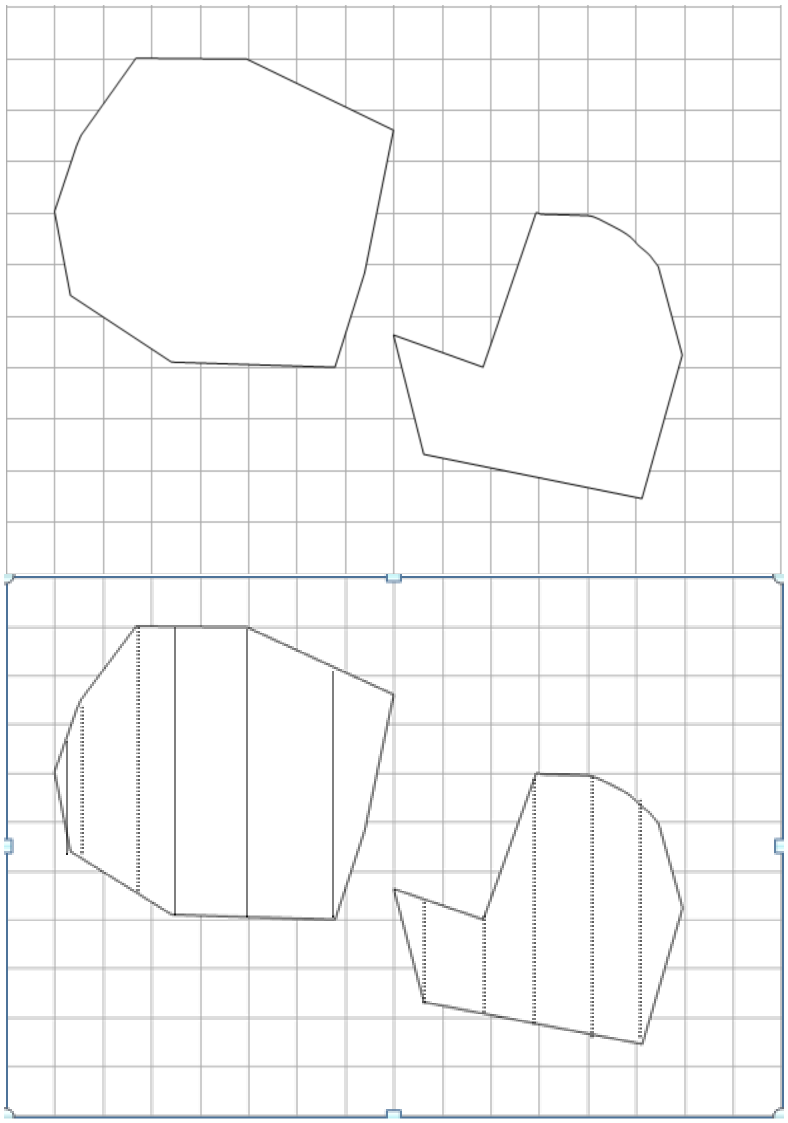


Figure 5: Shows polygons that have been decomposed to grids and trapezoids.

There are different kinds of decompositions based on types of primitive shapes created such as triangles or grids. It is very easy to create grids computationally inside a given environment but there are certain disadvantages associated with this scheme. The creation of grids for conducting decomposition of environment is a problem because it can contain grid elements which are partially filled when decomposing non grid-aligned polygons. Grid decomposition leaves out some portion in the single element of square which can lead to wasteful movement in the robot as shown with red grid elements in Figure 6 : Shows in the red region all the grid elements a robot cannot travel because of grid elements being partially filled. A strategy to mitigate this effect is to have smaller sized grid elements near the edges of non grid aligned polygons. The problems with this strategy are first, no matter how fine grained the grid becomes there will always be partially filled grid elements for non grid aligned edges of a polygon and second, a decrease in size of grid elements causes an increase in the finite transition system generated for the environment. Any increase in the finite transition system size will polynomially increase the cross-product automaton described in section 3.3 size created from the Büchi automaton and the discrete decomposed environment. This kind of increase in the state space would directly affect the computation and memory required in producing a motion plan.

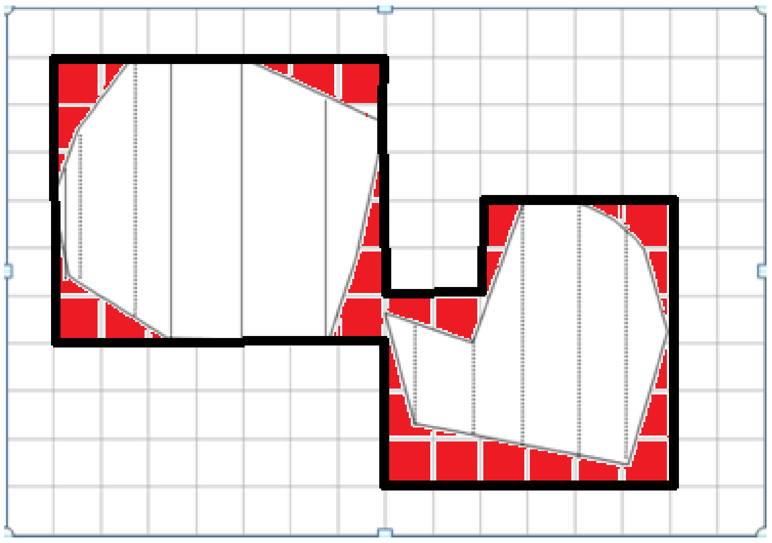


Figure 6 : Shows in the red region all the grid elements a robot cannot travel because of grid elements being partially filled.

Polygon decomposition has been pursued in many different directions besides grids, though creating decomposition with a minimum edge length criterion remains a hard problem to resolve. Specially in complex environment with convex polygons having holes in them, to get the minimum edge length means a NP-hard problem (Rourke & Supowith, 1983). We focus on decomposing polygons with holes in them because a normal environment will generally consist of a large structure enclosing smaller spaces which might be areas of interest themselves.

The other method is used for decomposing an environment is by decomposing from polygons into trapezoids as shown in Figure 5: Shows polygons that have been decomposed to grids and trapezoids. Trapezoids are further decomposed into triangles to have a better granularity. Triangulation as such strikes a balance between being able to create a decomposition which is computationally not very expensive, the total number of states produced by this method and granularity within the environment which would be fine to transit.

2.2 Similar research work

This section presents research that is similar to the one being created in this thesis project. Linear Temporal Logic Mission Planner (LTLMoP) is the one described first which provides a GUI to implement motion planning while using structured English as an input to control robots. The second one described here is LTL Robust Multi-Robot Planner (LROMP) is a software developed to plan optimal paths for a team of robots given high level specifications.

2.2.1 LTLMoP (Linear Temporal Logic Mission Planning)

This is a GUI based software developed at Cornell university which creates hybrid controllers synthesized from structured English to a motion plan for a robot. They have created an editor to modularize different components of motion planning and provide a way to work with them in isolation. The GUI also provides an interface for different types of robots whose properties can be modified.

There are three stages in which the entire processing takes place, described below (Finucane, Jing, & Kess-Gazit, 2010):

- Parse structured English into formal logic.
- Generate finite transition system from the formal logic created.
- Execute the automaton as a hybrid controller.

The LTLMoP consists of multiple editors namely, specification editor and region editor. The specification editor is where the structured English is provided as an input, with robot behavior defined by importing a robot description file. A region needs to be defined at this point either by the use of a region editor or by use of an imported region file. The parsing can finally be done once all these inputs are gathered. A successful parsing of the LTL specification will be indicated by the string "Automation Successfully Synthesized" in the output window while on failed parsing or LTL specifications being unrealizable nothing will be printed. If concrete counterparts to the region are available for the experimental environment, then the controller can be executed in the simulation or on a physical robot.

One of the biggest differences between the LTLMoP implementation and implementation described in this thesis is the interface with ROS. The ROS system enables access to a wider system of robots using packages developed by the open source community with few code changes. This greatly reduces the task of sending commands to a new robot being used in the physical world and perform the tasks. Another difference that we find between LTLMoP from this thesis implementation is the use of python as the programming language, which by the virtue of being an interpreted language will run slower than a compiled C++ executable and also have a larger memory footprint. The use of C++ also means it could potentially be used with real time operating systems which is not possible with a python implementation because of various factors such as garbage collection happening out of a users control.

2.2.2 LROMP(LTL Robust Optimal Multi-Robot Planner)

"This tool was developed for planning robust optimal paths for a team of robots subject to high level mission specifications." (Ulusoy, Smith, Ding, & Belta, 2012) Robot motion planning typically consists of reaching to a goal from an initial state while avoiding obstacles. LTL is defined for describing complex missions for robots. If we combine two concepts then it is possible to find an optimal path and feasibility for complex missions (Ulusoy, Smith, Ding, & Belta, 2012). Multi-robot path planning requires synchronization between different robots, which is taken care of by the use of timed automata (Ulusoy, Smith, Ding, & Belta, 2012).

The implementation done in LROMP is robust which means that it takes care of time drifts that can occur in real world because of change in physical characteristics of robots as compared to assumptions while creating simulations. The software consists of three major components:

- LROMP region automaton constructor(LROMP RAC) for generating behavior of the multi-robot system.
- Optimal-Run algorithm which finds an optimal run.
- Trace-closedness checker which ensures the given formula is trace closed.

The normal run while using this software would be for the user to define transitions for all each model robot and generate a product timed automaton from it using the LROMP RAC. The LTL formula is provided and checked using the trace-closedness checker about feasibility. This formula is finally fed to the Optimal-Run algorithm which finally provides the run for each individual robot. These runs can then be fed to actual robots and a real simulation can be run from them.

The difference between LROMP and current thesis implementation is, LROMP is targeted towards multiple robots and currently no environmental decomposition is available in LROMP. The environment is treated as a fixed graph which the robot visits instead of taking an arbitrary environment coordinate system and converting it into a graph. The system does not work on ROS which can provide it with a capability to interface with a new set of robot without much modification.

CHAPTER 3

ARCHITECTURE AND DESIGN OF SOFTWARE

This thesis tries to integrate different aspects of motion planning for an autonomous robot and create a single software package which takes specifications and environment as the input and produces a motion plan for the robot. To better understand the concept we explain the modules behind motion plans, the use of LTL in creating a motion plan and support libraries required for ensuring that all of this seamlessly comes together as a single software. The next page contains the architecture diagram.

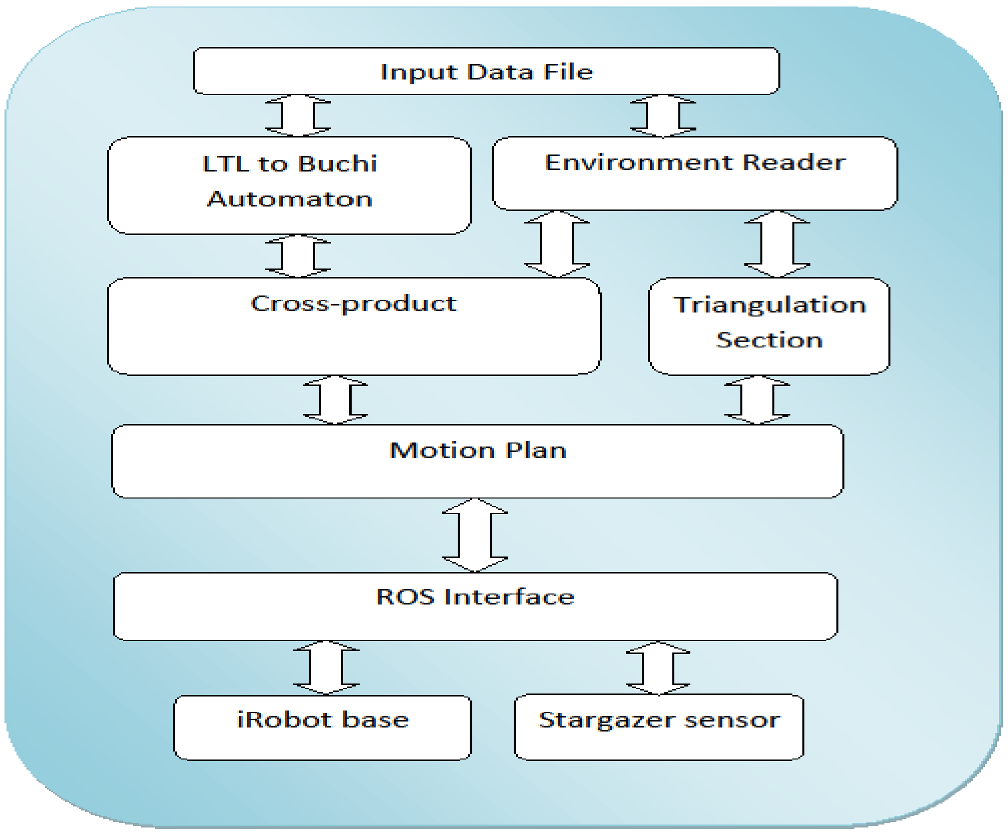


Figure 7: Architecture of software package

The Figure 7: Architecture of software package provides an overview of the first part of software system that has been implemented for this thesis project.

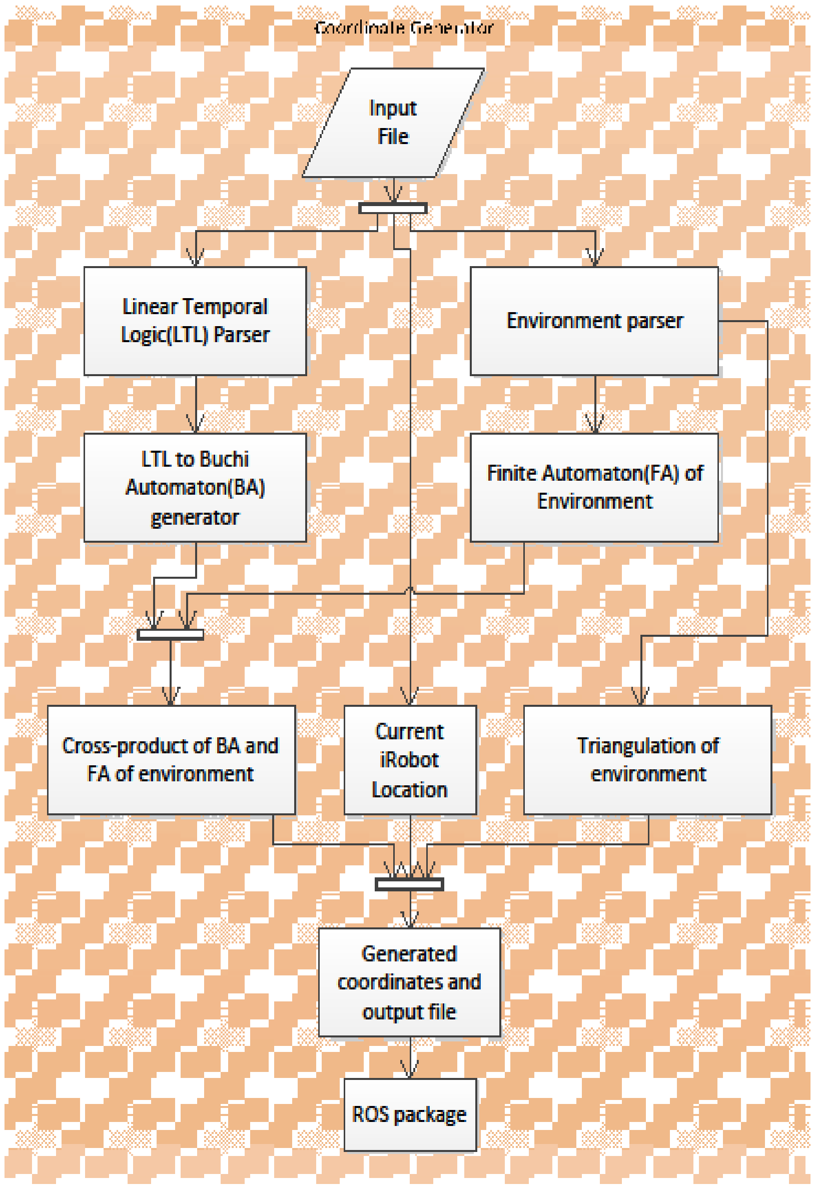

Coordinate Generator
Input File
Linear Temporal Logic(LTL) Parser
Environment parser
LTL to Buchi Automaton(BA) generator
Finite Automaton(FA) of Environment
Cross-product of BA and FA of environment
Current iRobot Location
Triangulation of environment
Generated coordinates and output file
ROS package

Figure 8: Design Diagram of Software package

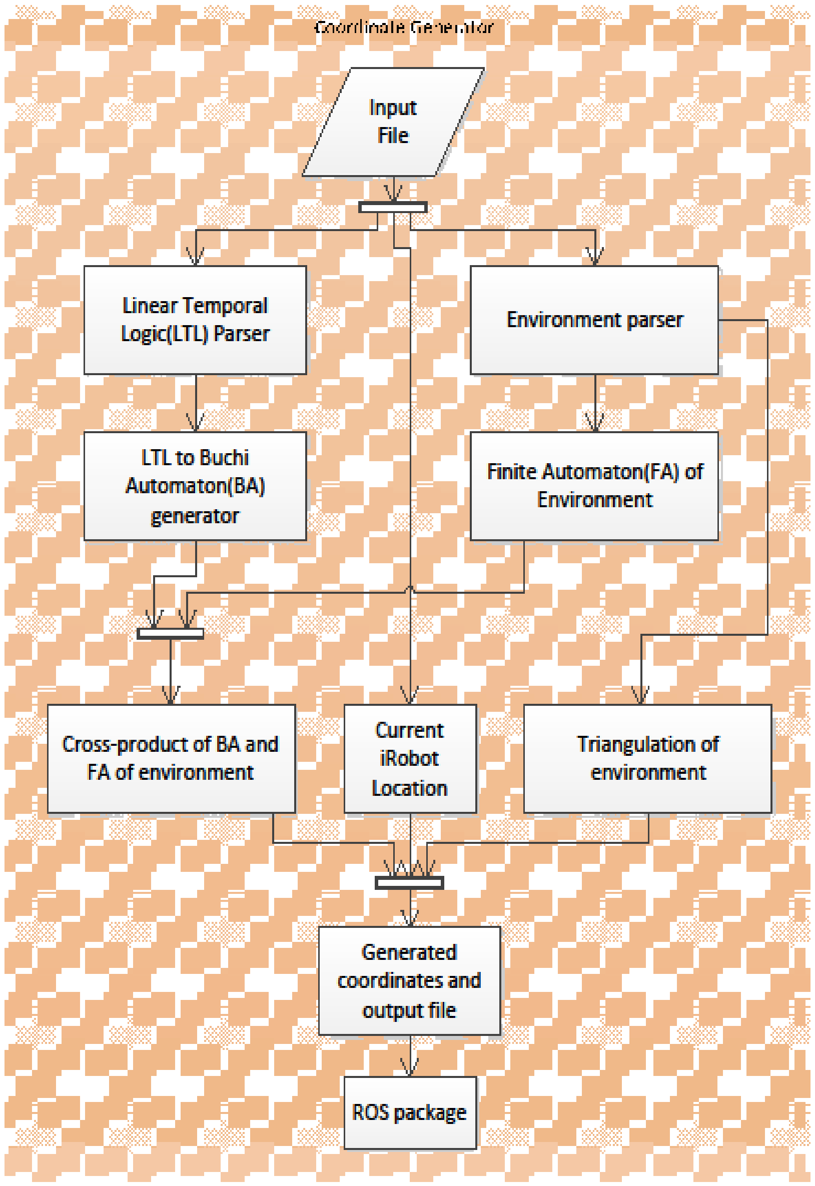

Coordinate Generator
Input File
Linear Temporal Logic(LTL) Parser
Environment parser
LTL to Buchi Automaton(BA) generator
Finite Automaton(FA) of Environment
Cross-product of BA and FA of environment
Current iRobot Location
Triangulation of environment
Generated coordinates and output file
ROS package

Figure 8: Design Diagram of Software package above shows the software design of the coordinate generator. It shows the major components involved in generating the final output of coordinates. The components are described below:

- LTL parser & LTL to BA generator: This block reads the data input and converts it into a string which is passed onto LTL and BA generator. The LTL to BA generator is a modified version of software developed by (Gastin & Oddoux, 2001). Described more in Section 3.1.

- Triangulation of environment: this block performs triangulation on all the polygons that have been defined in the environment. This is a modified software used from (Narkhede & Manocha, 1994). Further details are provided in Section 3.2.
- Cross-product of BA and FA of environment : the generation of cross-product and the resulting deterministic finite automaton is handled by this block of code. This block required the use of Binary Decision Diagrams(BDD) whose implementation can be found on (BDD library). The cross-product generation is described in further detail in Section 3.3.
- Visibility graph generation and Dijkstra algorithm: this block traverses through the finite automaton generated and uses Dijkstra's algorithm to find the shortest route possible for the robot to navigate. Further details are provided in Section 3.4.
- Steps for coordinate generation: Final steps on how the coordinates are generated for the lower planners to follow. Details present in Section 3.5.
- ROS Package: This is the package that works with a ROS system which then controls the robot to perform its tasks. Further details are provided in Section 3.6.

The following sections describe various components that were used to create the final software package that integrates LTL to BA, cross-product creation using BDD implementation, environment decomposition and triangulation, use of Dijkstera's algorithm for optimal motion plan and ROS package for robot movement. The algorithm implemented in this thesis is described in Figure 9: Algorithm used to describe the overall steps for implementation.

```
Overall algorithm for motion plan

Input. LTL specifications for robot motion ($\varphi$), Environment definition with labels and vertices (V), initial robot location ($l_0$), set of labels (labels).

Output. Set of coordinates for the iRobot to follow coord($\varphi$)

    1. $L_A \leftarrow$ Create label array (labels).
    2. $A_\varphi \leftarrow$ Create LTL to BA ($\varphi$).
    3. $E \leftarrow$ Create an environment model (V)
    4. $C_\varphi \leftarrow$ Create cross-product ($A_\varphi$ , E)
    5. $T_V \leftarrow$ environment decomposer (V, $L_A$ , E)
    6. $P_s \leftarrow$ starting polygon based on current position of robot.
    7. $C_{\varphi,initial} \leftarrow$ initial state of cross-product to use for finding execution trace.
    8. Execution_trace $\leftarrow$ bfs_crossprod($C_\varphi$ , E, $P_s$, $C_{\varphi,initial}$)
    9. coord($\varphi$) $\leftarrow$ post_polygon_coords(Execution_trace, $T_V$).
```

Figure 9: Algorithm used to describe the overall steps for implementation.

3.1 LTL to BA

In this thesis we use LTL to describe specifications which are used to generate models for the robot motion plan. An LTL specification could be equated to a safety property restricting the behavior of a system to stay within certain limits of execution. Such a restriction needs the LTL formula to be syntactically co-safe LTL formula which should only contain o, ◊ and U operators. A syntactically co-safe LTL can be converted into an automaton model called the Büchi automaton. A Büchi automaton is described as an automaton which can accept infinite inputs for a finite automaton (Kupferman & Vardi, 2001). Formally described as,

BA = (Q, $\sum$, δ, q, F)

where,

- Q : Set of states in the finite automaton
- $\sum$: set of symbols that form the language or in our case the labels associated with the BA
- $\delta : Q \times \sum \rightarrow Q$, is the transition function
- q : defines the initial state for the respective FA's.
- F : set of final states in the automaton.

In this thesis, we restrict ourselves to finite length motion planning problems which could be described using a special class of LTL formulas called co-safe LTL formulas. Any violation of a safety property for a system has to occur over a finite trace of execution. A LTL described in a Büchi automaton contains two parts, one is the prefix finite length part and the other is the lasso part having a loop in its execution trace for any non terminating system. Now from (Kupferman & Vardi, 2001) we infer that any safety LTL formula satisfying execution trace will have a finite good prefix. Classically to check if the model meets its specification ϕ we construct a Büchi Automaton $BA_{\neg\phi}$, for all traces violating ϕ followed by checking for feasible run between $BA_{\neg\phi}$ and the system model M. It is concluded by (Bhatia, Kavraki, & Vardi, 2010) that computationally it is better to find BA_{ϕ} and find a feasible trace with the system model M.

For the construction of a BA_{φ} the package created by Denis Oddoux and later modified by Paul Gastin serves the purpose of generating a Büchi Automaton from given LTL specifications. The software written is based on the method described in (Gastin & Oddoux, 2001). (Gastin & Oddoux, 2001) describes in detail about the efficient method proposed and implemented by the authors and its efficiency as compared to alternative implementation of Spin not being as quick in time or space complexity (Gastin & Oddoux, 2001). The improved versions of Spin did not produce substantially better results because the algorithms to generate the automatons remained the same as previous versions (Gastin & Oddoux, 2001). The core improvement that was proposed in this implementation which is also the reason for its efficiency is the conversion of LTL formulas to very weak alternating automatons which helps in reducing the number of states that need to be created in memory to "n", where n is less than the size of the formula. Typically a Büchi automaton would require the generation of $2^n * 2^n$ states, (Gastin & Oddoux, 2001) the authors create a very weak alternating automaton that is converted to a generalized Büchi automaton requiring at most 2^n states to be generated simultaneously. This generalized Büchi automaton is finally converted to a classical Büchi automaton which can be directly used. Another improvement that was brought on in this implementation was the simplification of the automaton at each step of Büchi automaton generation. This reduces the computational requirement of the next steps. In the end, these improvements contribute significantly to memory and time usage of the software code making it a suitable candidate for the purpose of this thesis. As motion planning is typically required to be quick, this software package provides a reliable and speedy method to extract Büchi automatons out of them.

The following Table 2 shows the comparison numbers for this particular implementation of LTL to BA using fast Büchi automaton generation method. The formula used to compare the efficiency was $\Phi_n = \neg(p_1 \; U(p_2 \; U \; (....U \; p_n)))$

Table 2: Comparison of runtimes for different packages used for LTL to BA conversion (Gastin & Oddoux, 2001)

No of propositions	Spin		Wring		EQLTL	LTL2BA	
	Time	Space	Time	Space	time	Time	Space
2	0.01	8	0.07	4100	8	0.01	3.2
3	0.03	110	0.29	4100	8	0.01	5.5
4	0.75	1700	1.34	4200	9	0.01	11
5	43	51000	10	4200	11	0.01	13
6	1200	920000	92	4500	15	0.15	25
7			720	6000	27	9.2	48
8					92	1200	93

For the purpose of this thesis a few modifications had to be made to this software to manipulate the Büchi automaton that was generated by the software described below. The output of the above referenced software is actually a Büchi automaton described in promela code. The final output was stored to a dynamically allocated array so that the usage could be better handled for further computations and an ease to manipulate the results. A set of links with labels were generated and added to this dynamic array to preserve the transition relation and the meaning of the code.

3.2 Environment decomposition and triangulation library

A continuous environment needs to be decomposed into discrete states so that it could be treated as a set of finite states (Kuan, Zamiska, & Brooks, 1985). The reason for such a decomposition is for the robot being able to travel inside the environment without colliding with an obstacle. The method chosen for decomposition is splitting the environment into polygons further decomposed into smaller primitives. These algorithms have classically been computationally expensive processes where polygons are decomposed into smaller primitives, up until algorithms were proposed by (Seidel) which was much easier to implement as compared to the algorithms previously proposed (Seidel). Seidel proposed an algorithm producing results in O(n log* n) time complexity, where (log* n) means when n>0, log* n denotes the largest integer l so that $\log^{(l)} n \geq 1$ and $\log^{(i)}n = \log(\log^{(i-1)}n)$, i>0, $\log^{(0)} n = n$. This algorithm was implemented by the team of two researchers (Narkhede & Manocha, 1994). Their implementation has been put in the public domain which is used by this project where the environment consists of a set of simple polygons. The output from the previously mentioned software is a set of triangles whose centroids can be used as a point to travel within the environment without colliding into obstacles. The implementation is able to quickly generate results for a simple polygon environment which then produces the finite state machine to create the requirement.

The mechanism for the triangulation algorithm is explained further here. It is a type of vertical decomposition. The software supports the presence of holes inside polygons which are recursively triangulated. The triangulation works in 3 phases as explained by (Narkhede & Manocha, 1994) on their site. The steps are as follows:

1. Decompose the polygon into trapezoids by adding line segments which are non-horizontal and non-intersecting line segments in the polygons defined.

2. Decompose the trapezoids into monotone polygons which are computed from the trapezoidal decomposition of the polygons.
3. "Triangulate the monotone polygons by cutting off convex corners of convex polygons." (Narkhede & Manocha, 1994) This can be performed in linear time.

The memory requirements of this algorithm are based on the number of vertices of a polygon and number of holes present in the polygon. The allocation of memory has to happen before running the algorithm based on the environment data that has been passed by the user, hence a dynamic allocation is performed for all the required memory. This allocated memory is then passed onto the software component to perform the triangulation and return the results.

Table 3: Runtimes of polygon decomposition over 100 iterations (Narkhede & Manocha,1994)

Number of Vertices	Running Time
10	0.9 ms
50	3.5 ms
100	6.7 ms
500	42.7 ms
1000	97.6 ms
5000	590.0 ms
10000	1.24 s
50000	7.3 s
100000	15.45 s

The Table 3: Runtimes of polygon decomposition over 100 iterations (Narkhede & Manocha,1994) above shows measurements taken over a HP series 735 with run time averages. These numbers show that even with complicated environment and a large number of vertices the software can efficiently manage a workload and produce results quickly. For motion planning of mobile robots it becomes a priority to generate the results in time.

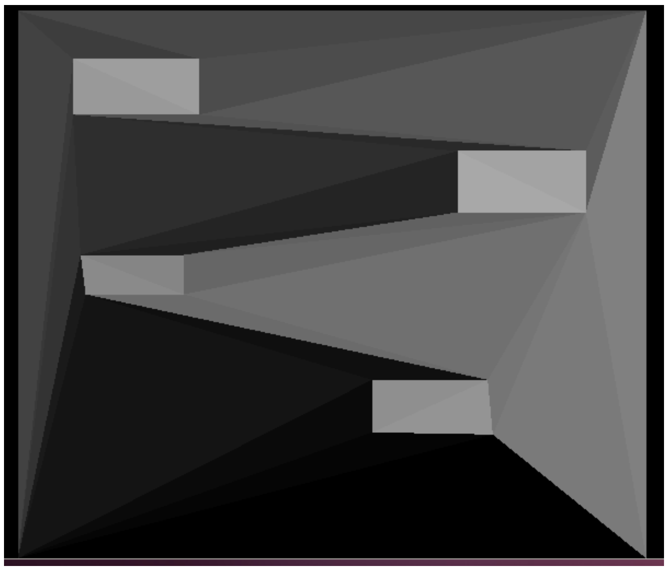


Figure 10: Shows visualization of triangulated polygons

The Figure 10: Shows visualization of triangulated polygons above shows a visualization of triangulation of an environment. Even the polygons which act as holes for the master polygons have been triangulated. Modifications had to be made to the input and output mechanism for data. The final set of triangles that are created by the use of this algorithm are stored in an array to be used later for traversal. An adjacency list is also created to calculate which triangle share an edge with another where triangle T_1 and triangle T_2 are adjacent to each other only when T_1, T_2 share a common line segment with a relation $\rightarrow_D$ for system defined here.

3.3 Binary Decision Diagrams library and cross product generation

Binary decision diagram (BDD) is a data structure that is extensively used for model checking and synthesis of hardware circuits. BDD's are actually compact encoding of splitting trees used to evaluate Boolean expressions. A BDD has a few properties (1) being an acyclic graph, (2) it has shared sub-trees and (3) presents a compact way to represent Boolean functions. These trees are ordered such that if a branch goes from node i to node j then i < j, the BDD also has to be in a reduced form, these two properties if adhered to define a BDD. The following Figure 11: Depicting the splitting tree of $(q \rightarrow p) \wedge r \rightarrow (p \leftrightarrow r) \wedge q$ and Figure 12: Depicting the binary decision diagram of $(q \rightarrow p) \wedge r \rightarrow (p \leftrightarrow r) \wedge q$ show the Boolean expression $(q \rightarrow p) \wedge r \rightarrow (p \leftrightarrow r) \wedge q$. It clearly depicts how easily a decision tree can be compressed using a BDD and how compact the final representation becomes. This property of BDD is helpful in finding the satisfiability of Boolean expressions because it has lesser memory and computation requirements to parse from root to sink of the tree.

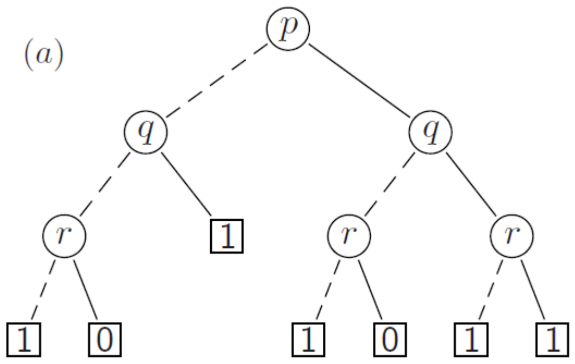


Figure 11: Depicting the splitting tree of $(q \rightarrow p) \wedge r \rightarrow (p \leftrightarrow r) \wedge q$

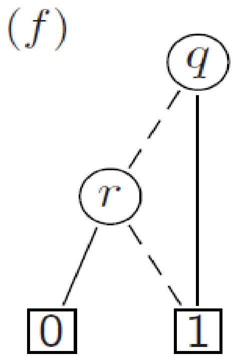


Figure 12: Depicting the binary decision diagram of (q → p) ^ r → (p ↔ r) ^ q

A library has been created by Carnegie Melon University (BDD library) which can evaluate Boolean expressions by the use of BDD and check their satisfiability. This library enables the creation of cross-product between the finite state machine generated from the environment and the Büchi automaton that was generated from the LTL specifications.

Cross-products are created by evaluating Boolean expressions quickly and being able to produce results regarding their truth values. The Boolean expressions are created by the use of labels associated with environment and BA finite state machines transitions. Each transition in both the state machines consist of two components, the state they are pointing to and a transition with a Boolean expression. Creating a cross-product of the two mentioned finite state machines or finite automaton (FA) means there needs to be a finite state machine which can accept intersection of two finite automatons $FA_1 \cap FA_2$. Generating an intersection of two finite automaton requires being able to evaluate the conjunction of the transition functions. This conjunction requires Boolean functions to be evaluated. We evaluate and keep the transitions based on satisfiability of the Boolean function as an output by the BDD. Checking satisfiability can become exponentially large which makes it very important for the program to perform efficiently. A BDD representation in this case provides an opportunity to help in completing this step quickly. Here we include the formal definition of the two FA's followed by the definition of cross-product FA. The definitions are as follows:

$FA_1 = (Q_1, \sum, \delta_1, q_{0,1}, F_1)$

$FA_2 = (Q_2, \sum, \delta_2, q_{0,2}, F_2)$

where,

- Q_1 , Q_2 = Set of states in the finite automatons
- $\sum$ = set of symbols that form the language or in our case the labels associated with the FA
- δ_1 , δ_2 = these are the transition functions that define the Boolean expression that needs to be satisfied to take a transition
- $q_{0,1}$, $q_{0,2}$ = defines the initial state for the respective FA's.
- F_1 , F_2 = defines the set of accepting state present in the FA's.

As we are using environment as one of the FA we make a few assumptions on the system of FA created from the environment FA. The final state can be any state in the environment which can be accessed by the robot and the robot's initial state can be any of the states in the environment as long as the robot can reach that state. There is also an assumption of bidirectional accessibility in case of environment while creating the FA. Making these assumptions make the accepting states of the BA in combination with any state in the environment as a final state in the cross-product. It means that final state will be dictated essentially by the BA that is being used. Also the set of initial states will be restricted by the current position of the robot and the initial state defined in the BA. The cross-product output will be constructed in the following form

$FA_{final} = (Q_1 \times Q_2, \sum, \delta, (q_{0,1}, q_{0,2}), F_1 \times F_2)$

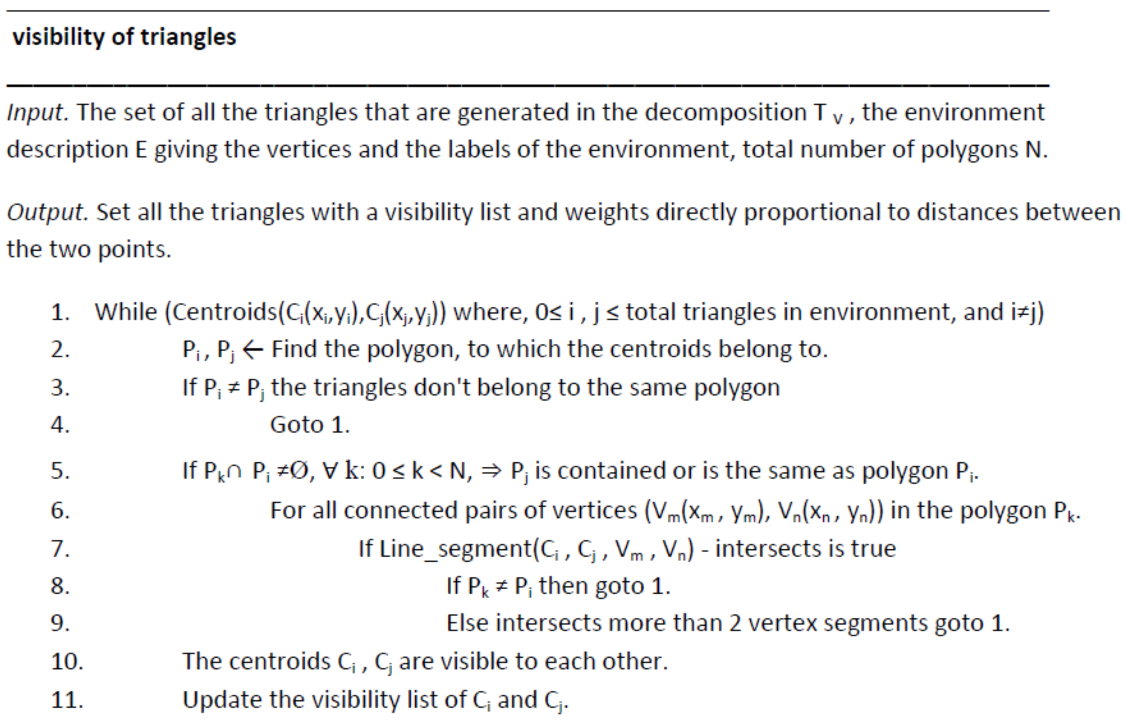
where,

- $Q_1 \times Q_2$ = combination of all the states in the two FA's
- $\sum$ = set of symbols that form the language or in our case the labels associated with the FA
- $\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
- (q_0^1, q_0^2) = this is the combination of the initial state from BA and all the states in the environment.
- $F_1 \times F_2$ = it represents the final state which is essentially the combination of all the states in the environment FA and the accepting states in BA.

Hence BDD library is used to form the transition function while creating the cross-product and serves an important role in generating the cross-product. As the number of states are combinatorial in nature the process of evaluating Boolean functions needs to be quick. This cross-product is then used to generate the motion plan for the robot based on the initial value it gets as the starting point.

3.4 Visibility graph generation and Dijkstra algorithm

Visibility graph is constructed for set of points and obstacles in Euclidean plane (Berg, Cheong, Kreveld, & Overmars, 2008). Generation of the visibility graph between centroids of the triangle is done by taking pairs of centroids in the same polygon and verifying their line of sight with respect to each centroid against any obstacles. Obstacles in this case are defined as any object or line segment that could block a clear line of sight between two centroids, it could either be an opaque polygon that lies within the parent polygon containing triangles of interest or it could be an edge of the parent polygon itself which is intersecting the line of sight for the current pair of centroids. In either case an obstacle if found implies a direct line segment cannot connect the pair of centroids. The steps for calculating the visibility are shown in Figure 13 : Algorithm for constructing visibility graph..



visibility of triangles

Input. The set of all the triangles that are generated in the decomposition T $_V$, the environment description E giving the vertices and the labels of the environment, total number of polygons N.

Output. Set all the triangles with a visibility list and weights directly proportional to distances between the two points.

```
1.   While (Centroids($C_i(x_i,y_i)$,$C_j(x_j,y_j)$) where, $0 \le i , j \le$ total triangles in environment, and $i \ne j$)
2.           $P_i , P_j \leftarrow$ Find the polygon, to which the centroids belong to.
3.           If $P_i \ne P_j$ the triangles don't belong to the same polygon
4.                   Goto 1.
5.           If $P_k \cap P_i \ne \emptyset$, $\forall$ k: $0 \le k < N$, $\Rightarrow$ $P_j$ is contained or is the same as polygon $P_i$.
6.                   For all connected pairs of vertices ($V_m(x_m , y_m)$, $V_n(x_n , y_n)$) in the polygon $P_k$.
7.                           If Line_segment($C_i$ , $C_j$ , $V_m$ , $V_n$) - intersects is true
8.                                   If $P_k \ne P_i$ then goto 1.
9.                                   Else intersects more than 2 vertex segments goto 1.
10.          The centroids $C_i$ , $C_j$ are visible to each other.
11.          Update the visibility list of $C_i$ and $C_j$.
```

Figure 13 : Algorithm for constructing visibility graph.

The calculation for deriving the visibility is provided in more details, by determining the coordinates of interest first, in the current context they are centroids of two triangles within the same polygon. Let's call them c1 and c2 with coordinates being x1,y1 and x2,y2 respectively. Using Euclidean geometry we derive a linear equation between these coordinates in the following form:

$$y = \left(\frac{y1-y1}{x1-x2}\right) * x + \left(y1 - \left(\frac{y1-y2}{x1-x2}\right) * x1\right) \ ,$$

we further normalize this equation to the form

$$b1 * y = a1 * x + c1$$

The next part is to check for visibility between centroids where we verify against all the edges of the parent polygon containing both the triangles and all the edges for child polygons contained inside of the parent polygon. The edges are evaluated in a clockwise fashion to generate individual lines between each vertex of the polygon. Let us say that we take two vertices of one of the polygon that we are going to check against for visibility v1 and v2 with the following coordinates x3,y3 and x4,y4 respectively. We derive another linear equation using these coordinates in the form above.

$$b2 * y = a2 * x + c2$$

Now we try to find a point of contact between these two linear equations by solving them. On solving the two equations for x and y coordinates we find the point of intersection between the two lines.

$$x = \frac{(b2 * c1) - (c2 * b1)}{(a2 * b1) - (a1 * b2)}$$

$$y = \frac{(a2 * c1) - (c2 * a1)}{(a2 * b1) - (a1 * b2)}$$

Where, $b1 \neq 0$ && $((a2 * b1) - (a1 * b2)) \neq 0$

The next challenge is to ensure these lines intersect within the areas of interest. The area of interest is defined by the lines intersecting each other within the line segments created by centroids c1, c2 and v1, v2 coordinates. As long as the point of intersection lies outside these line segments the two centroids fall in each other's line of sight as shown in the Figure 14: Shows area of interest in the visibility graph generation.Using this formula we derive the visibility graph and find all pairs of centroids connected to each other directly adding the weights on each of the transitions based on their distances from each other.

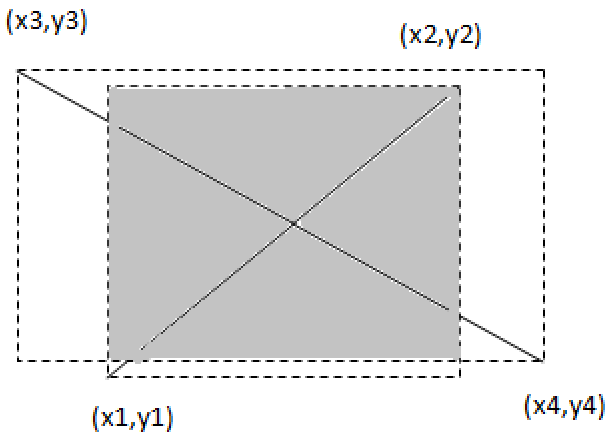


Figure 14: Shows area of interest in the visibility graph generation.

Using these guides we continue with our calculation of the entire array of triangles for each individual polygon and pair of centroids. Once we have created the visibility graph we run a Dijkstra algorithm in order to find the shortest path possible. The Dijkstra algorithm takes into consideration the weight on each transition which is calculated on the basis of the distance between centroids in meters. On each step the nodes are evaluated and the shortest path is calculated from these nodes. While populating the transition nodes of the system we also calculate the weight on each transition. The weight of each transition is the distance between the pair of centroids of the triangles.

We run the Dijkstra algorithm and find the path which has the shortest weight/distance between the triangles being posed as source and destination. The visibility graph is build only on a polygon to polygon basis, which means only those centroids will be connected to each other which are in line of sight and belong to the same polygon. This condition has been chosen because the LTL specifications can be arbitrary and can choose different polygons to reach the accepting state. Hence, we build the visibility graph such that it stays within the confines of the same state in effect guaranteeing the satisfiability. It means that we are guaranteed by the use of this visibility graph to not move into a different polygon which might violate some other specification.

The next problem is being able to generate a path plan between polygons which will help the robot transit from one polygon to another, as a visibility graph can only ensure reaching from one point in the polygon to another point within the same polygon, not transiting between polygons. This is resolved by finding the closest centroids between the two polygons i.e. find the centroid closest to the destination in one polygon to the centroid in the other polygon. By finding the closest centroids we can create a direct transition between the polygons. There are two potential problems, first if the triangulation results in very thin triangles it could eventually lead to conditions where the robot could transition over the robots and cross obstacles that should not have been visited by the robot due to geometry. The completion system previously mentioned also leads to the problem of possibly not having the shortest path, because the closest centroids of two different polygons could be located anywhere with respect to current robot location. Hence although the robot might be at one corner of the destination polygon it might have to navigate around the same destination polygon to reach the final centroid. This problem cannot easily be solved as it requires the visibility graph to be redrawn with respect to the robot and from there calculations have to be done for the shortest path. This can be a computationally expensive task if the number of polygons to be visited is large. So in order to maintain a good performance on the amount of computations required to generate the motion we do not recalculate the visibility graph again.

3.5 Steps for coordinate generation

The generation of a final motion plan is composed of a few high level steps which can be further broken down into smaller tasks. The following steps produce the coordinate to coordinate motion plan for the robot:

1. After the creation of a cross-product as described in previous sections we can find different polygons that need to be visited in the system. It means using cross-product states we determine polygons to be visited by the motion planning system.
2. This parsing of cross-product is done by using breadth first search that tries to use as few state transitions as possible. This algorithm has been chosen so that the number of polygons to be traversed are minimized. Although before traversing through the cross-product we need to find the current location of the robot. This is because there needs to be a starting point within the cross-product which is then taken as the starting state for parsing through the current initial state.
3. Once the polygon transitions are obtained from the traversal of cross-product we find the closest points to travel within the polygons. This is determined by the use of a function called nearest_poly_triangle() which based on the polygon being considered and the current location tries to locate a triangle which will be the

closest to it. It returns the index of the closest triangle of the specific polygon only closest to the current position. The triangle whose index has been returned is then used as the destination. We continue this process till we have transitioned through all the possible states in the cross-product path.

4. The next step is to be able to travel through all these polygons based on the triangles that have been generated from the polygon areas. While generating the polygons we create a visibility graph as described earlier.
5. We run Dijkstra algorithm on the visibility graph and create paths which should be close to optimal paths within the polygon while ensuring the satisfiability of the specifications. This is done by travelling in shortest paths within the polygons and then directly transitioning to the next polygon.
6. The coordinates of the centroids visited are provided as the motion path plan to the robot to follow inside the environment. These form the final step in coordinate generation which is transferred to the ROS package for the actual locomotion.

3.6 ROSCPP and ROS framework

" ROS is an open source, meta-operating system for robot. It provides services such as hardware abstraction and message passing." (ROS wikipedia, 2013). ROS framework tries to keep itself to a minimum and provides the infrastructure required for achieving compatibility between different types of systems whether it is different languages or hardware. The goal is to keep the interfaces simple and be able to integrate different types of packages which can then easily be modified and used for a variety of applications.

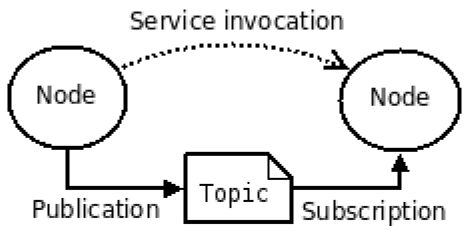


Figure 15: Shows the Publisher and Subscriber relation in ROS (ROS Wikipedia, 2013)

The core of ROS is to provide a distributed computing environment which has independent processes running. These processes communicate via ROS messaging services and make it easier to be run on multiple platforms. There are three conceptual levels at which ROS works namely (ROS wikipedia, 2013):

1. **File system level**: At this level we see a general ordering of the software that is created under ROS. ROS helps in creating an organized hierarchical software environment where group of packages called stacks help in creating the collaborative environment. These packages serve as the basic unit for software which generally accomplishes a single functional task for e.g. getting data from a sensor and providing it in a more general format like temperature in Fahrenheit , controlling robot movement based on higher level commands such as left and right instead of exact differential inputs.
2. **Computation graph level**: ROS at this level concerns with the running of a client-server architecture at an abstract level where a ROS master process acts as a central server directing all the messaging traffic and nodes. Nodes are independent processes running as clients in a distributed environment. These nodes interact using messaging services which could to one-to-one or many-to-one. These messages could also be stored by using a concept called bags for

archiving data. A topic acts as the point of access for information exchange, nodes can subscribe to a topic and publish to a topic without knowing who is reading or sending information to them. This decouples a node from another and keeps the software architecture as decoupled as possible.

3. **Community level**: As ROS tries to embody as much distributed architecture it can, it also promotes different communities to be able to exchange their software packages more easily. It makes it easier for different groups or teams to exchange packages which are built independently but serve a purpose which can solve a specific problem for e.g. an iRobot package was easily integrated into the current system without having any need to modify its contents. As with any open source software there is a need to establish community support which can answer questions posed by users and generate a healthy discussion to achieve their goals. This kind of support is provided by ROS wiki, Bug ticket system and ROS answers.

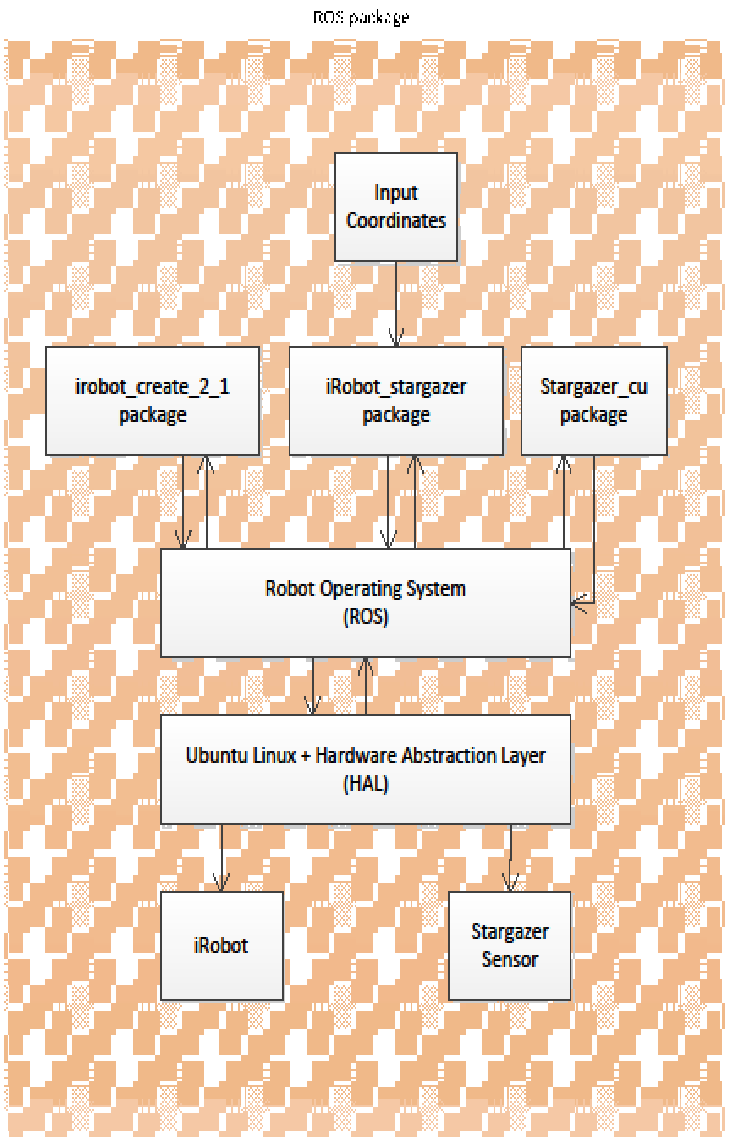
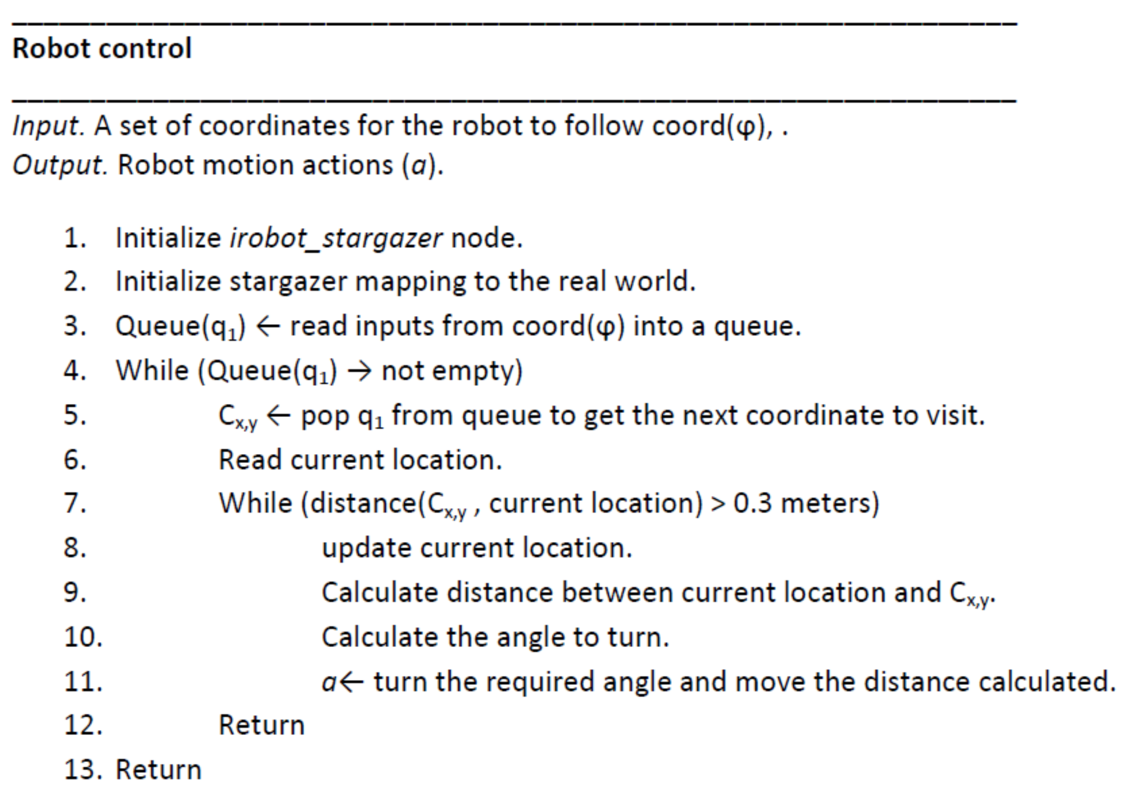

ROS package
Input Coordinates
irobot_create_2_1 package
iRobot_stargazer package
Stargazer_cu package
Robot Operating System (ROS)
Ubuntu Linux + Hardware Abstraction Layer (HAL)
iRobot
Stargazer Sensor

Figure 16: irobot_stargazer package design diagram.

Client libraries are created to ease the job of being able to make ROS concepts accessible through code (ROS wikipedia, 2013). ROSCPP is a client library that enables the creation of ROS topics, services and parameters (ROS wikipedia, 2013). This library is extensively used by C++ programmers to create packages which can be used to generate backend code freeing the developer to concentrate on their specific tasks instead of writing the interfaces with ROS nodes. It also comes with some standard set of messages that can be used to access data from a variety of sensors, localization and sending control messages to robots.

The steps described in Figure 17: Algorithm steps for lower planner to run the robot based on coordinates generated from the higher planner. Describe the implementation.



Robot control

```
Input. A set of coordinates for the robot to follow coord(φ), .
Output. Robot motion actions (a).

1.  Initialize irobot_stargazer node.
2.  Initialize stargazer mapping to the real world.
3.  Queue(q_1) ← read inputs from coord(φ) into a queue.
4.  While (Queue(q_1) → not empty)
5.          C_{x,y} ← pop q_1 from queue to get the next coordinate to visit.
6.          Read current location.
7.          While (distance(C_{x,y} , current location) > 0.3 meters)
8.                  update current location.
9.                  Calculate distance between current location and C_{x,y}.
10.                 Calculate the angle to turn.
11.                 a← turn the required angle and move the distance calculated.
12.         Return
13. Return
```

Figure 17: Algorithm steps for lower planner to run the robot based on coordinates generated from the higher planner.

The package created for ROS is currently named irobot_stargazer because of its use of iRobot Create and Stargazer hardware. The irobot_stargazer package is used to convert all the coordinate information into actual robot motion. This package reads the output file from the coordinate generator executable and is based on the information it receives from the sensors directs the robot towards its destination. The irobot_stargazer package works in the following manner:

- It initializes a ROS node with the name as irobot_stargazer.
- It subscribes to the stargazer_cu node to gather localization information from the stargazer sensor.
- It initializes the stargazer map which gives locations of the landmark coordinates in the real world.
- The ROS package loads all the coordinates into a coordinates queue which are used as navigation points for the robot.
- An infinite loop is created in another thread which will monitor coordinates queue, acquire data from the stargazer sensor and give commands to the iRobot create. This creates the closed loop interface between reading of data and being able to execute functions on the robot.

CHAPTER 4

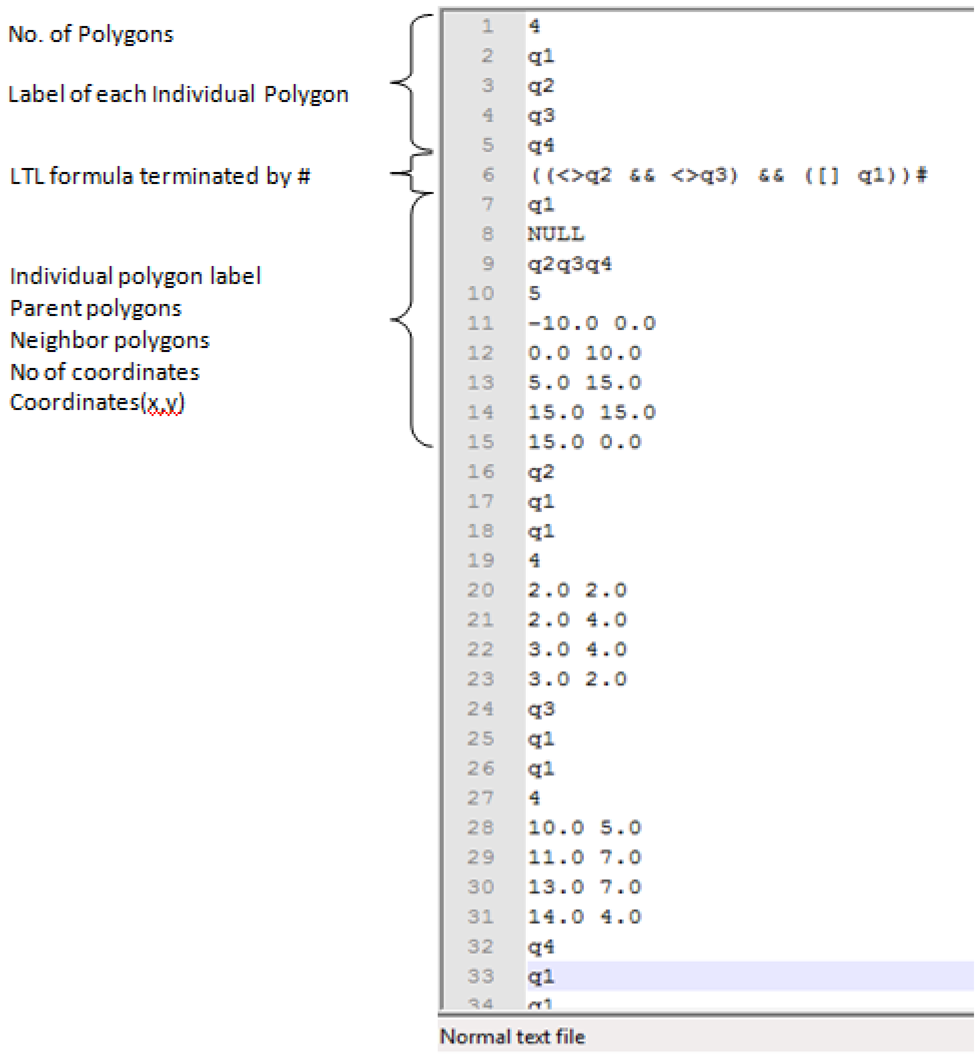
INPUT AND OUTPUT DATA FILES

The input file has the following fields that are required for the processing to take place. The fields are described below:

- Number of atomic propositions in the LTL
- List of atomic propositions
- LTL formula
- Description of each polygon consists of following points[1]:
    - Atomic symbol denoting the polygon
    - Concatenated atomic proposition of polygons in which above named polygon resides
    - Concatenated atomic propositions of all the neighbors to the polygon.
    - Number of vertices in the polygon.
    - List of coordinates specified in meters in float format "x" coordinate followed by "y" coordinate.

The Figure 18 below shows an example of a data file which contains all the required entities.

[1] Enables nesting of multiple polygons by giving information on parent and neighbor polygons.



```
4
q1
q2
q3
q4
((<>q2 && <>q3) && ([] q1))#
q1
NULL
q2q3q4
5
-10.0 0.0
0.0 10.0
5.0 15.0
15.0 15.0
15.0 0.0
q2
q1
q1
4
2.0 2.0
2.0 4.0
3.0 4.0
3.0 2.0
q3
q1
q1
4
10.0 5.0
11.0 7.0
13.0 7.0
14.0 4.0
q4
q1
q1
```

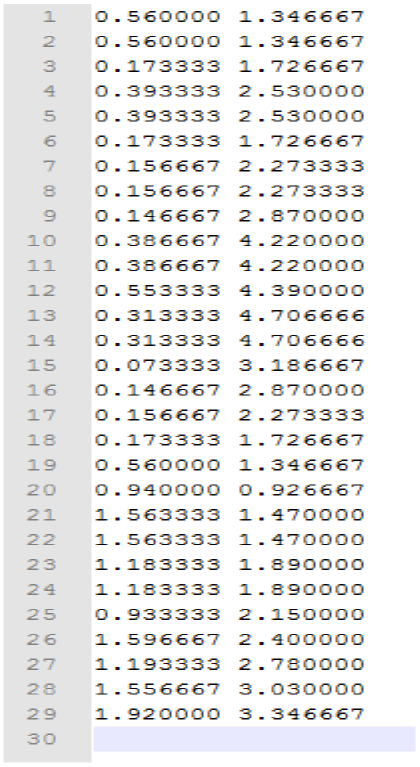
Figure 18: Input data file.

Following points need to be noted:

- Each atomic proposition corresponds to a polygon in the environment hence, the number of polygons and atomic propositions have to be equal.
- The definition of neighbor is any polygon that is adjacent to another polygon such that the robot can travel between the two polygons without having to move into any other third polygon.

- The units of coordinates are meters and correspond to the real world coordinate so it can generate accurate coordinates for navigation by the system.
- The LTL string has to terminate with "#" as to indicate the end of formula.
- Parents of a polygon are also considered as their neighbors because transition from the parent polygon to the child polygon can be done without having to transit through another polygon in between.
- Each child polygon is considered to be entirely contained inside of the parent polygon because of restrictions posed in the triangulation algorithm.
- For the polygon which contains all the other polygons and does not have a parent, a label value which does not contain any know atomic proposition like "NULL" can be chosen.
- No spaces are allowed in the concatenated sections of the label data while defining the polygons parents or neighbors.
- The polygons are supposed to have 4 or more vertices because of restrictions posed by triangulation algorithms.
- The polygons should be simple closed polygons that may contain holes in them otherwise triangulation fails in case of self intersecting polygons.
- Vertices should not be repeated when defining polygons for the environment.

A file called coordinates is generated as the output by this software which specifies the centroids of the triangles the robot should navigate to. The coordinates file acts as an input to the ROS package which in turn drives the robot to the goal. This file is very simple in its interpretation and contains x and y coordinates sequentially that the robot is going to travel to based on the initial starting point as provided to the software. The Figure 19: Coordinate output. below shows the screenshot of the output file generated by the software tool. This output was generated from the example file shown in the Figure 18 above.



```
0.560000 1.346667
0.560000 1.346667
0.173333 1.726667
0.393333 2.530000
0.393333 2.530000
0.173333 1.726667
0.156667 2.273333
0.156667 2.273333
0.146667 2.870000
0.386667 4.220000
0.386667 4.220000
0.553333 4.390000
0.313333 4.706666
0.313333 4.706666
0.073333 3.186667
0.146667 2.870000
0.156667 2.273333
0.173333 1.726667
0.560000 1.346667
0.940000 0.926667
1.563333 1.470000
1.563333 1.470000
1.183333 1.890000
1.183333 1.890000
0.933333 2.150000
1.596667 2.400000
1.193333 2.780000
1.556667 3.030000
1.920000 3.346667

```

Figure 19: Coordinate output.

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1 Experimental setup

Experiments were done using an iRobot in the Cyber Physical Systems lab with the same example data as shown in figures in Chapter 4. Below Figure 20: Displaying the robot environment on conceptual and physical level shows the simulated environment and the physical environment created in the Cyber Physical Systems Lab.

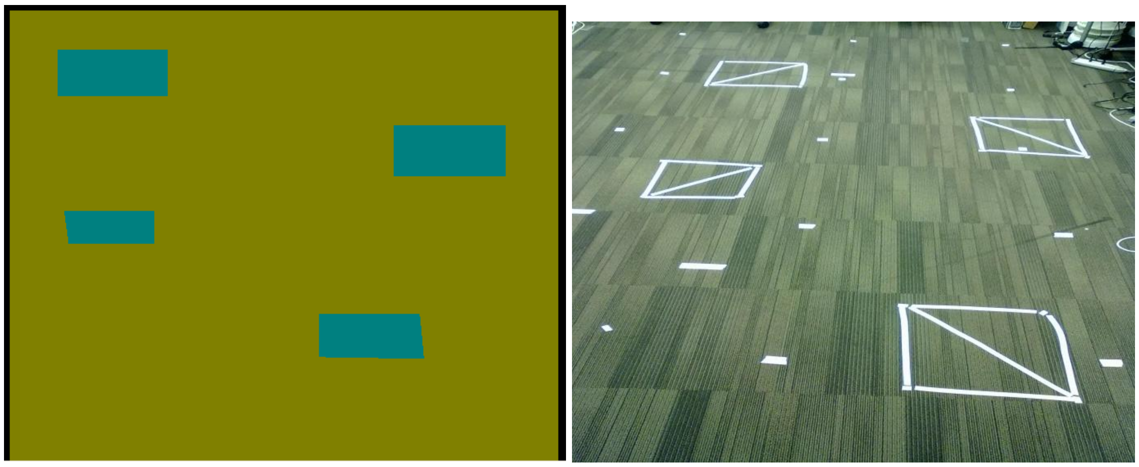


Figure 20: Displaying the robot environment on conceptual and physical level

The environment is defined as labels q_i, where i = 1,..5 naming each polygon. The outermost polygon was named as q_1. Three experiments were conducted using the iRobot and stargazer sensor in which they covered the following requirements of LTL.

- Coverage: the robot covers all the polygons defined in the system with no particular order being imposed. The LTL to express this was (([]q1) && (<> q2 && <>q3 && <> q4 && <>q5))

- Sequencing: the robot is asked to cover polygons in a defined order of sequence for e.g. visit q1 followed by q2 and so on. LTL used for this experiment was (([]q1) && (<> (q2 && <>(q4 && <> (q3 && <>q5)))))
- Reachability with avoidance: the robot is able to avoid polygons which are marked as such. The LTL used for this experiment was (([]q1) && (<> q5 && <>q3 && <> q4) &&(!q3 U q2)).

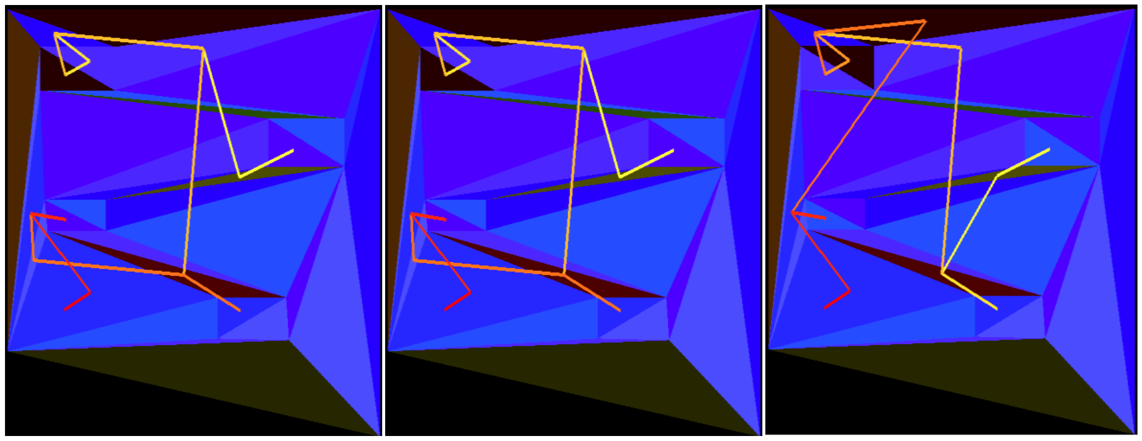


Figure 21: shows reachability, coverage, sequencing

The experiments were successful and showed that the robot fulfilled the LTL goals of navigating through the environment as specified. The final goals were achieved using the ROS package.

5.2 Results

The thesis describes about the software that was developed for motion planning for a robot with navigation points and then a ROS package which has the ability to actually navigate the robot. It is a modular system which creates motion plans for robots based on specifications from LTL. The features of the software are as follows:

- Parse LTL specifications and generate a Büchi Automaton which is stored into a form that can be manipulated easily.
- Parse the environment into a finite automaton and decompose the environment into triangles which can then be used to generate the motion plan for the robot.
- Create the cross-product between the Büchi automaton and finite automaton of environment which is used to generate the transitions of states that can are used to satisfy the LTL under constraints from the environment.
- The coordinates generated as output by the software are then plugged into the ROS package irobot_stargazer which produces the actual robot control.

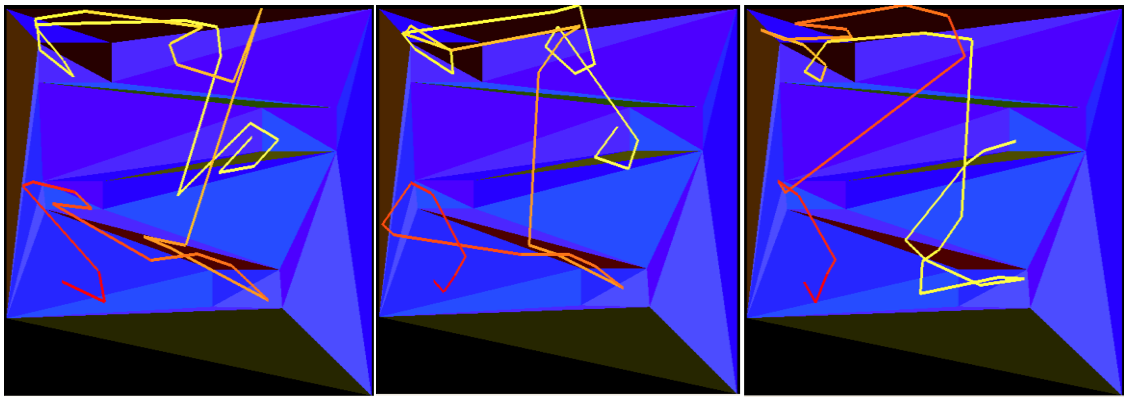


Figure 22 : Shows Reachability, Coverage and Sequencing

CHAPTER 6

FUTURE WORK

While completing this thesis some future work has been identified which can be put as improvements on the current package. The following features that can add further value to the tool:

- Provide an option to the user to decompose the environment into either triangles or trapezoids instead of only triangles.
- Move from a 2 dimensional model to a 3 dimensional model and have a more generic motion planner.
- There could also be the addition of multiple robots to the system in which the motion plan is centrally generated and sent out individually to robot though that would require a substantial change to the current software.
- Currently we assume that the LTL formula provided has a reachable accepting state which will finally produce the result. The aforementioned scenario can be averted by the use of automaton relaxation and get the user to change the formula based on feedback provided by the software. (Kim & Fainekos, 2012).

BIBLIOGRAPHY

BDD library. (n.d.). Retrieved 2013, from http://www.cs.cmu.edu/~modelcheck/bdd.html

Belta, C., Isler, V., & Pappas, G. J. (2005). Discrete Abstractions for Robot Motion Planning and Control in Polygon Environments. *IEEE Transactions on Robotics*, (pp. 864-874).

Berg, M. d., Cheong, O., Kreveld, M. v., & Overmars, M. (2008). *Computation Geometry: Algorithms and Applications.* Springer.

Bhatia, A., Kavraki, L. E., & Vardi, M. Y. (2010). Sampling-based Motion Planning with Temporal Goals. *International Conference on Robotics and Automation* (pp. 2689-2696). Anchorage: IEEE.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model Checking.* MIT press.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* , 269-271.

Dijkstra's. (n.d.). *Dijkstra's algorithm*. Retrieved March 2013, from Wikipedia: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dzifcak, J., Scheutz, M., Baral, C., & Schermerhorn, P. (2009). What to do and how to do it: Translating Natural Language Directives into Temporal and Dynamic Logic Represenation for Goal Management and Action Execution. *IEEE conference on robotics and automation.* IEEE.

Fainekos, G. E., Kress-Gazit, H., & Pappas, G. J. (2005). Hybrid controllers for path planning: A temporal logic approach. *44th IEEE Conference on Decision and Control*, (pp. 4885-4890).

Fainekos, G. E., Kress-Gazit, H., & Pappas, G. J. (2005). Temporal logic motion planning for mobile robots. *IEEE International Conference on Robotics*, (pp. 2020-2025).

Fainekos, G. E., Kress-Gazit, H., & Pappas, G. J. (2005). Temporal Logic Motion Planning for Mobile Robots. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation.* Barcelona, Spain.

Finucane, C., Jing, G., & Kess-Gazit, H. (2010). LTLMoP: Experimenting with language, Temporal Logic and robot control. *Intelligent Robots and Systems* (pp. 1988-1993). New York,USA: IEEE/RSJ.

Gastin, P., & Oddoux, D. (2001). Fast LTL to Buchi Automata Translation. *13th International Conference on Computer Aided Verification*, (pp. 53-65). Paris, France.

iRobot Create owners manual. (n.d.). Retrieved 2013, from http://www.irobot.com/filelibrary/create/create%20manual_final.pdf

Keil, J. M. (1996). *Polygon Decomostion.*

Kim, K., & Fainekos, G. E. (2012). Approximate Solutions for the Minimal Revision Problem of Specification Automata. *IROS.* Vilamoura: IEEE/RSJ.

Kim, K., Fainekos, G. E., & Sankarnarayanan, S. (2012). On the revision problem of specification automata.

Knuth, D. *Art of Programming vol 3.*

Knuth, D. (2010). *Art of Programming vol 4.*

Kress-Gazit, H., & Pappas, G. J. (2010). Automatic synthesis of robot controllers for tasks with locative prepositions. *IEEE international conference on robotics and automation.*

Kress-Gazit, H., Fainekos, G. E., & Pappas, G. J. (2009). Temporal Logic-based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics* (pp. 1370-1381). IEEE.

Kroger, F., & Merz, S. *Temporal logic and state systems.* Springer.

Kuan, D. T., Zamiska, J. C., & Brooks, R. A. (1985). Natural decomposition of free space for path planning. *ICRA.*

Kupferman, O., & Vardi, M. Y. (2001). Formal methods in System Design. In *Formal methods in System Design* (pp. 291-314).

Lamport, L. (1980). "Sometime" is sometimes "not never". *ACM* , 174-185.

LaValle, S. M. (2006). *Planning Algorithms.* Cambridge University Press.

Nahar, S., & Sahni, S. (1988). Fast Algorithm for Polygon Decomposition. *IEEE Transactions on computer-aided design vol 7* (pp. 473-483). IEEE.

Narkhede, A., & Manocha, D. (1994). *Fast polygon triangulation algorithm based on Seidel's algorithm.* Chapel Hill: UNC-CH.

Pnueli, A. (1977). The Temporal Logic of Programs. *Foundation of Computer Science, 1977, 18th annual symposium,* (pp. 46-57).

ROS wikipedia. (2013). Retrieved 2013, from http://ros.org/wiki

Rourke, J. O., & Supowith, K. J. (1983, March). Some NP-Hard Ploygon Decomposition Problems. *IEEE Transactions on Information Theory* , p. 181.

Seidel, R. A simple and fast incremental randomized algorithm for computing trapezoidal decomposition and for triangulating polygons. In *Computational Geometry: Theory and Applications* (pp. 51-64).

Stargazer User manual. (n.d.). Retrieved 2013, from http://www.hagisonic.com/

Su, S.-K. (2012). *Model-Based Development of Multi-iRobot Simulation and Control.* Tempe: Arizona State University.

Ulusoy, A., Smith, S. L., Ding, X. C., & Belta, C. (2012). Robust Multi-Robot Optimal Path Planning with Temporal Logic Constraints. *ICRA - IEEE international conference*, (pp. 4693-4698).

Vik, S. (2013). *An Implementation of a Near-Linear Polygon Triangulation Algorithm for General Polygons*. Retrieved 2013, from Home of Sigbjorn Vik: http://sigbjorn.vik.name/

APPENDIX A

SOFTWARE AND HARDWARE SETUP

A.1 Software and hardware setup

The following requirements are needed to run this software package:

- Ubuntu Linux 10.14 and higher.
- Gcc compiler 4.2 and above.
- ROS and its support software. For more details check (ROS wikipedia, 2013).
- ROS irobot package from Brown university and its support software.
- ROS stargazer_cu package and its support software from university of Colorado.
- A computing system with 2 USB ports at least, to communicate with the hardware.
- Install the irobot_stargazer package and the coordinate_generation software developed specifically for this thesis project.
- Stargazer needs to be mounted centrally on the iRobot with their axis aligned to each other.

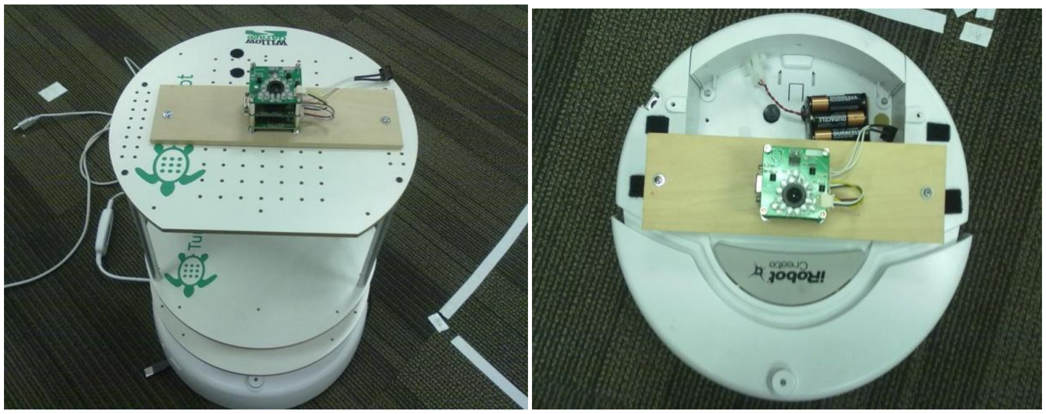


Figure 23: Showing two configurations of iRobot with Hagisonic Stargazer mounted

A.2 Execution Setup

The setup required to run the software is as follows:

1. ROS core should be running in a terminal if it isn't configured to initialize automatically.
2. Turn on the iRobot create and stargazer sensor and connect them to the system using separate USB ports. It should be noted that correct ports are passed onto each package while initializing the respective packages.
3. The packages need to be running and configured at this point by giving the following commands in separate terminals, "rosrun irobot_create_2_1 driver.py" and "rosrun stargazer_cu stargazer_cu".
4. To check that both packages are functioning correctly the following commands can be typed into a new terminal, "rostopic echo /cu/stargazer_marker_cu" will display the readouts from stargazer sensor with distance and the landmark id in it. To check the initialization of irobot_create_2_1 these commands "rosservice call tank 0 100 100" followed by "rosservice call tank 0 0 0" should be executed in a new terminal.
5. Once it is confirmed that stargazer_cu and irobot_create_2_1 packages are running as expected we need to open another terminal to run data file containing the LTL through coordinate_generation. This creates a file called coordinates.txt.
6. Bring up the ROS package irobot_stargazer to read the file and execute the motion plan. The robot starts moving after reading this file and executes navigation maneuvers.

7. The iRobot create will execute motion while getting localization data from the stargazer sensor. Finally it should reach its expected destination and come to a halt.

APPENDIX B

HARDWARE USED FOR THE EXPERIMENTS

B.1 iRobot Create

The robot which provides the locomotion is called an iRobot. This robot has been used because of ease of interface available to control it through ROS. This hardware also has a wide array of sensors and uses differential motors for its movement. The serial port is a RS-232 which can be connected to a keyspan USB converter and connected to a laptop or any other device supporting USB connections.

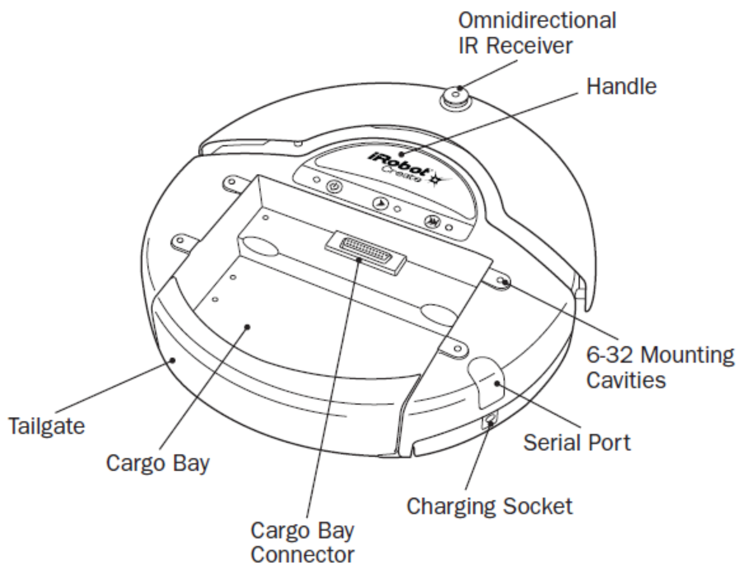


Figure 24: Top view of iRobot with components. (iRobot Create owner's manual)

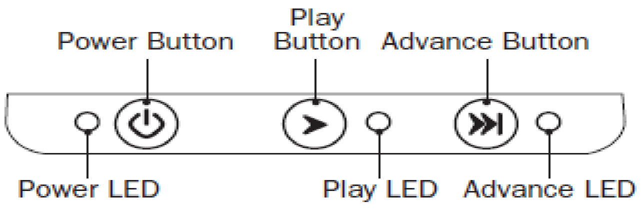


Figure 25: Showing the button panel of iRobot create. (iRobot Create owners manual)

The Figure 24 & 25 above shows the button panel where power button is used to boot up the iRobot create and start communicating over the serial port. The iRobot also has internal demo modes which can be triggered by the use of play button. The Figure 26 below shows underside of iRobot create.

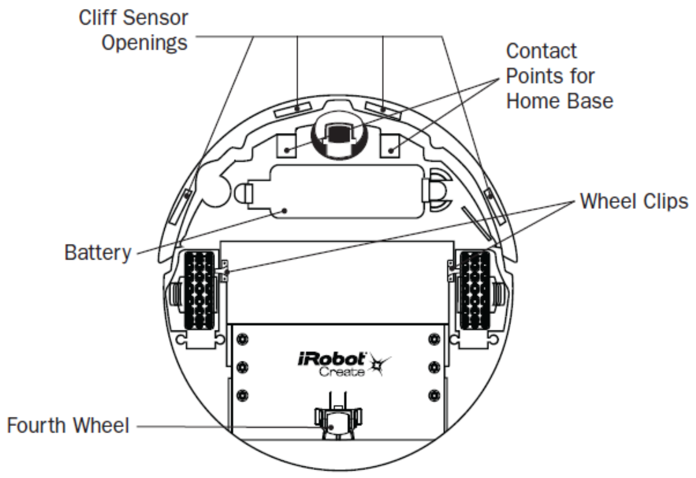


Figure 26: Shows the underside of iRobot (iRobot Create owner's manual)

where we can locate the battery which is charged through the charging socket. The point of interests are specifically the two differential motors attached to the wheel. These two wheels provide the drive to iRobot and can be controlled by the use of irobot_create_2_1 package in ROS. The commands passed on the serial queue to be executed are described in detail (iRobot Create owners manual).

B.2 Hagisonic Stargazer

This is a sensor created by the company Hagisonic called stargazer. It is a unique sensor system in which indoor localization of a robot can easily be achieved by analyzing reflected rays of Infra-red from passive landmarks attached to the ceiling. It also makes these markers high resolution in determining their current location based on the sensors orientation with reference to passive markers. (Stargazer User manual)

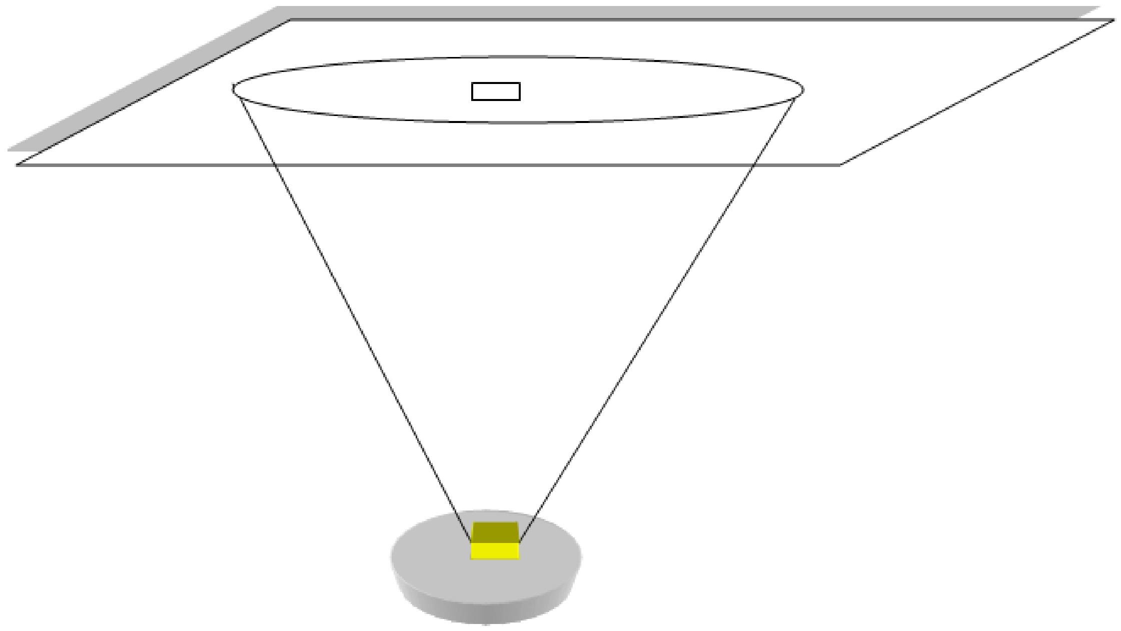


Figure 27 : Shows stargazer mounted on an iRobot create seeing a landmark

The components required for the stargazer setup are:

- StarGazer sensor module with DSP module, IRED Module, Support, Lens Hood.
- Connector cable, serial and 3-pin connector.
- Power supply module, power circuit and adapter.
- Passive landmarks.

Remarks: The sensor although claimed to be unaffected by normal reflective surfaces showed behavior that indicated otherwise. A map needs to be created for the location of markers in the real world to localize the robot in the environment. The sensor itself provides capability to perform this task by using a map mode though as indicated in (Su, 2012), it was better to use alone mode and create a map with manual measurements for deriving current location. The stargazer sensors were configured in alone mode with the unit of measurement being Feet for distance and Degrees for angle.

B.3 Stargazer Map

The following Figure 28 shows the stargazer map that was used for the thesis project where it defines the work area. The map area is same one as defined in (Su, 2012). . The entire Cyber Physical Systems lab work area had been covered by these landmarks attached to the false ceilings.

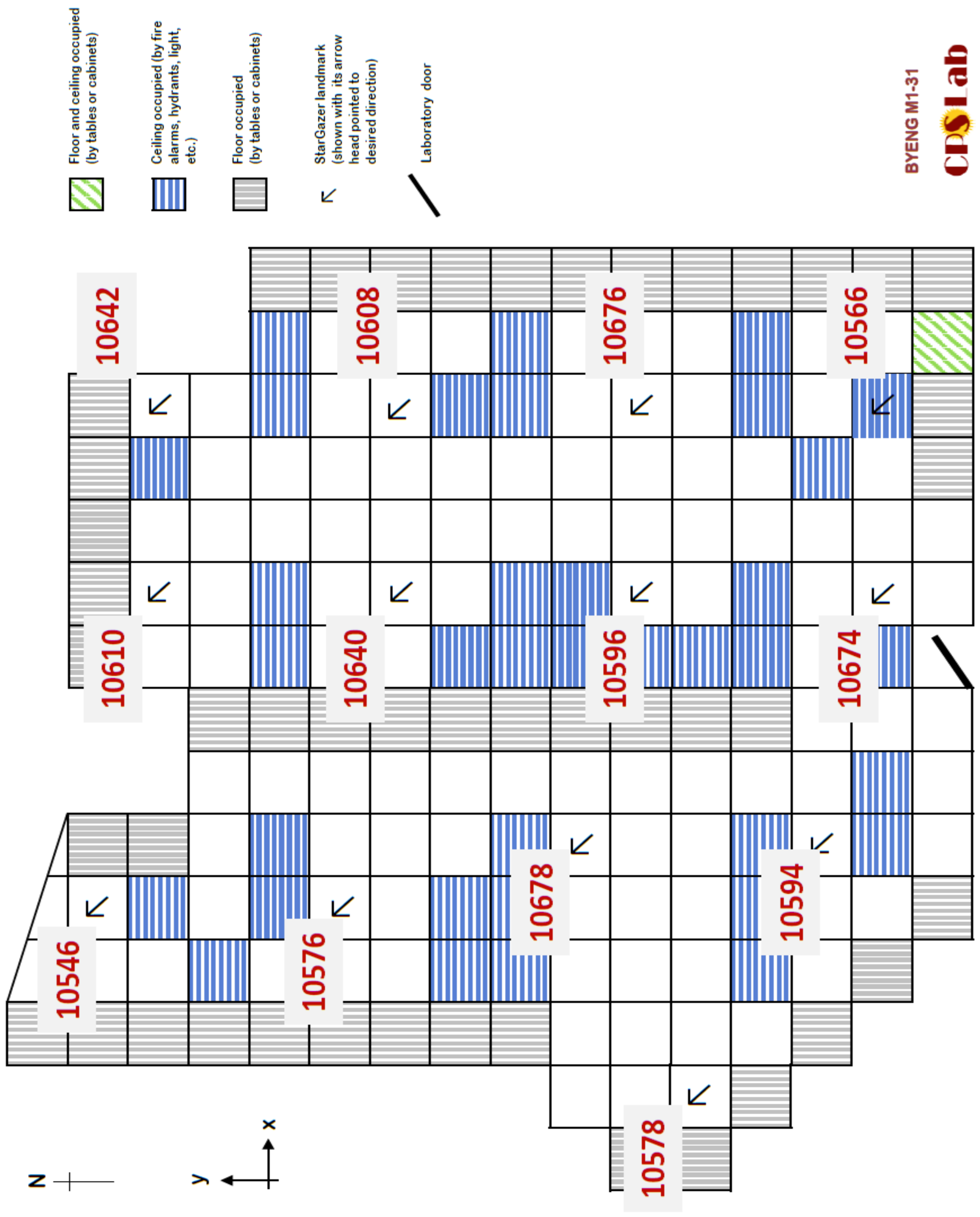

Floor and ceiling occupied (by tables or cabinets)
Ceiling occupied (by fire alarms, hydrants, light, etc.)
Floor occupied (by tables or cabinets)
StarGazer landmark (shown with its arrow head pointed to desired direction)
Laboratory door
BYENG M1-31
CPSLab
10642
10608
10676
10566
10610
10640
10596
10674
10546
10576
10678
10594
10578
N
x
y

Figure 28: Stargazer map for cyber physical systems lab