University of Kaiserslautern
Department of Computer Science

Master's Thesis

Reactive Controller Synthesis for Mobile Robotics

Author:
Adrian Leva

Supervisors:
Ph.D. Rupak Majumdar
Prof. Dr. Karsten Berns

Technical Report MPI-SWS-2017-001

Abstract

Often linear temporal logic is used to describe the desired behaviour of a robot. Once the specification or environment is complex enough, generating a controller based on that specification can be time intensive. In order to approach this problem, Schmuck and Majumdar have introduced a formalism and algorithm to model and synthesize specifications in [23] by introducing a hierarchy over local specifications. In this work we have adapted that algorithm, so it can be run in a simple simulation as well as in a realistic 3D simulation to control a robot. We could conclude that the time to generate the controller for the robot can be cut down by using this approach. Running it in a realistic scenario with unforeseen events such as obstacles that were not defined beforehand, we ran into additional problems. Therefore we extended the algorithm to try to resolve the problems in those situations.

Zusammenfassung

Oft wird lineare temporale Logik benutzt um das gewünschte Verhalten eines Roboters zu beschreiben. Sofern diese Spezifikation oder die Umgebung aufwändig wird, kann es sein, dass das Generieren eines Controllers für den Roboter sehr lange dauert. Um dieses Problem zu lösen haben Schmuck und Majumdar in [23] einen Formalismus und einen Algorithmus um eine Spezifikation zu modellieren und synthetisieren, indem sie eine Hierarchie über lokale Spezifikationen einführen. In dieser Arbeit haben wir diesen Algorithmus angepasst, sodass er sowohl in einer einfachen Simulation als auch in einer realistischen 3D simluation genutzt werden kann um einen Roboter zu steuern. Zudem konnten wir feststellen, dass mit diesem Ansatz Zeit bei der Erstellung des Controllers gespart werden kann. Beim Ausführen in einer realistischeren Umgebung mit unvorhergesehen Ereignissen, wie etwa Hindernissen, die vorher nicht spezifiziert waren, traten zusätzliche Probleme auf. Deshalb haben wir den Algorithmus erweitert um in solchen Situation die Probleme zu lösen.

Contents

1. Introduction

Handling robots and giving them tasks can be very complex. As one approach, specifications the robot has to satisfy can be given as *linear temporal logic* (LTL) formulas, as tasks like reachability or sequences of goals can be naturally expressed in LTL.

However, once the specification is complex and the state space of the scenario is getting big enough, synthesizing a controller from that specification can take a long time — it does not scale well. This could render the synthesis impractical, if these tasks change over time or are assigned on demand. Consider a warehouse, where robots should fetch items when they are ordered. If getting the item takes less time than generating the controller to actually execute that task, taking this approach of controller synthesis is infeasible.

As one approach to solve the scalability issue, Schmuck and Majumdar suggested a hierarchical approach in [23]. It leverages the fact that many scenarios can be divided into smaller subareas, that are connected and introducing a hierarchy that represents the game on different abstraction levels. We show this at an example consisting of multiple buildings that are refined by the rooms inside.

Thus, over the course of this thesis, we present the approach and describe how we have adapted this algorithm to be able to run it in a simple simulation on top of the *Linear Temporal Logic MissiOn Planning* (LTLMoP) [8] toolkit, where the robot is displayed as a dot and the environment is a 2D map.

LTLMoP's architecture allows different robot configurations, so building the algorithm around it enables us to run it on various robots in simulation. We could also show, that depending on the situation, time can be saved by using this approach of hierarchical synthesis, by using a smart partitioning.

On limited hardware resources, which can be the case if the synthesis is done on the robot itself, it even enabled us to complete the synthesis, where without our adaption the procedure would run out of memory, due to its size. We have compared that time needed to generate the controller in the buildings example of the hierarchical algorithm with the hierarchical one and could see a time improvement in most cases.

To take it one step further we have integrated that with a framework to run on real robots, namely the *Robot Operating System* (ROS) [10] and the realistic 3D simulator *Gazebo* [9]. A more realistic environment is more complex and unforeseen situations can happen, like the robot is unable to move because of

an obstacle. We have addressed some of these additional problems, by further changing the algorithm to be able to react in such cases. Because it allows us to compute a strategy in relatively short time, we can change the specification to add additional constraints while the robot is in operation without too much downtime. By only adding additional constraints or changing the order of goals, the rest of the original specification is still in place, so the robot will still stay safe as much as it is possible with some inaccuracy or stop if it cannot complete the specification.

If this kind of safety is too stringent, another motion planning algorithm can be used, such as Dijkstra, to definitely find a way to a goal, even if it means the robot moves through regions the synthesis did not consider. The robot is then unlikely to fail due to obstacles, but this approach should not be used if the specification defines regions the robot should avoid.

But not only in the case of unforeseen situations the specification can be changed. Should some part of the specification change, the existing strategy can be deleted and the robot will generate the new one the next time it will need it, or it could be done manually, while the robot is still running. As an example, the robot was cleaning only the hallways and the cafeteria, but now it should clean the office as well. While it is still cleaning, the office could be added to the specification without the need to restart the robot.

To conclude the work, we discuss our findings and give an outlook for improvements that still could be done, such as improving the usability of the tools to incorporate the hierarchical approach or learning about permanent obstacles and thus changing the specification automatically and permanently as well, based on these observations.

2. Related Work

Synthesis of specifications in LTL are well-studied in general. For exampe Fainekos et al. have shown how temporal logic motion planning of more higher level specifications, such as visiting multiple goals while avoiding dangerous regions or visiting goals in a specific order, can be used [7].

Hierarchies have been introduced in many areas, not only controller synthesis, such as a bottom-up approach by Aminof, Mogavero and Murano in [1] to synthesize a system by building components from already existing ones, to create more complex systems.

Related to the integration of controller synthesis into the Robot Operating System, Wong, Finucane and Kress-Gazit have shown that it is possible in [24].

In [19] Ramaithitima et al. have tried to overcome the drawback of exponential growth of the environment when represented as graph by introducing an abstraction. In order to do so they divide the configuration space into partitions of topological similar configurations. Instead of specifications based on temporal logic, they studied pursuit-evasion problems, where multipe pursuers have to detect evaders.

Kloetzer, Ding and Belta [15] suggested an approach to several unicycles that should fulfill a global specification. They split the problem into a hierarchy, by first abstracting the motion of the robots, then composing these abstraction, generating a motion for the team and finally transforming these abstractions into controls for the individual robots.

Belkhouche and Belkhouche [2] introduced a hierarchy in their motion planner as well. To describe the system, they use differential equations for the continuous state and graphs for the discrete part. The hierarchy is reflected by four modes for the planner, move-to-goal, move-to-inter-goal, obstacle-avoidance and path smoothing. Given some goal the robot starts in the move-to-goal mode until it detects some obstacle in its way. It switches to the move-to-inter-goal to move close to the obstacle and then changes to the obstacle-avoidance mode. The path smoothing is activated whenever the robot switches between other modes.

Saha et al. [22] have used an satisfiability modulo theories (SMT) solver to solve the motion planning problem for multiple robots by first building a library of motion primitives and then composing these to a complete trajectory.

However, at the time of writing we have not found an attempt to combine such a hierarchical approach to controller synthesis with a realistic robotics sim-

ulation.

3. Preliminaries

3.1. Game Solving and GR(1)

In order to describe the behaviour of the robot and possibly the environment, a notation has to be established, and one way to do so is by using Linear Temporal Logic (LTL). Consider an example, where we have three regions, 1 and 2 representing buildings that are connected by some region 3, as seen in Figure 3.1.

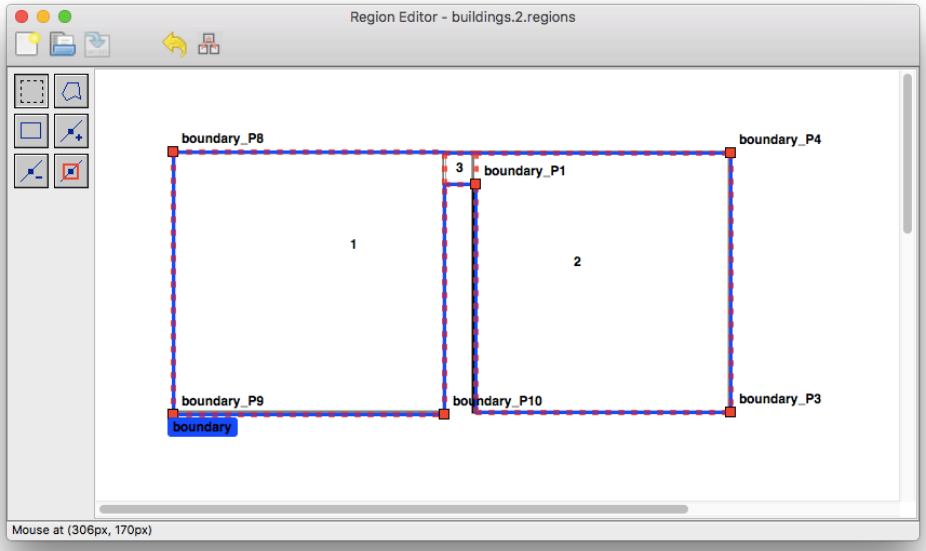


Figure 3.1.: LTLMoP's region editor with the example

The robot should patrol between those two buildings, so the specification could be stated as *go to 1; go to 2*. This specification has to be expressed as LTL formula, since we want to use an algorithm that works on LTL to solve it.

LTL formulas describe an infinite sequence of states. There is a finite number of atomic propositions, basic logical operators like negation ($\neg$) and or ($\vee$) as well as temporal operators like next ($\bigcirc$) and until ($\mathcal{U}$). Thus the basic syntax is

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi$$

where π is an atomic proposition. Intuitively $\bigcirc \varphi$ means that φ has to hold in the next state and with $\varphi \mathcal{U} \psi$, φ has to hold in every state until a state is reached, where ψ is true. From these basic operations other important ones like globally $\Box \varphi$ and eventually $\Diamond \varphi$, defined as $\Diamond \varphi = \top \mathcal{U} \varphi$ and $\Box \varphi = \neg \Diamond \neg \varphi$. Following these definitions, eventually means the formula has to hold at some point, whereas globally expresses that it has to hold in every state of the sequence.

Going back to our example of buildings the robot should patrol between, the specification for the robot could be written as $\Box \Diamond 1 \wedge \Box \Diamond 2$, meaning that the robot should visit region 1 and region 2 infinitely often.

The realizability of LTL is 2-EXPTIME-complete [21], which makes it practically infeasible. For a subset of LTL formulas, namely Generalized Reactivity (GR(1)) [17] formulas, there is an algorithm that decides realizability in N^3. These formulas are of the form

$$\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s \,, \tag{3.1}$$

with φ^e describing the environment or sensor propositions and φ^s modeling the system or robot. φ_i^e and φ_i^s are nontemporal Boolean formulas that represent the initial state of the environment and system respectively. φ_t^e is of the form $\bigwedge_{i \in I} \Box B_i$, where B_i are Boolean combinations of system and environment proposition as well as the temporal next operator with some environment proposition, to describe the behaviour of the environment. Similarly, φ_t^s describes the behaviour of the system, with the difference that it can rely on the current system state as well as the current and next sensor outputs, since it first updates these and then takes an action. Finally, φ_g describes the goals of the environment and system, in the form of $\Box \Diamond B_i$, where B_i can be environment or system propositions. If we apply that to our example we get this:

$$\begin{aligned} &(\top \wedge \Box \top \wedge \Box \Diamond \top) \rightarrow \\ &\qquad ((1 \vee 2 \vee 3) \\ &\qquad \wedge (\Box(1 \rightarrow \bigcirc 3) \wedge \Box(2 \rightarrow \bigcirc 3) \wedge \Box(3 \rightarrow \bigcirc(2 \vee 1))) \\ &\qquad \wedge (\Box \Diamond 1 \wedge \Box \Diamond 2)) \end{aligned} \tag{3.2}$$

The propositions are just the labels of the regions, 1, 2 and 3. Furthermore, the assumptions of the environment are all true, since we have none. The behaviour of the robot is stating that it starts in either region, captures the transitions between the regions and finally describes the goals.

Synthesis of that specification can then be done with the help of GR(1) games [17]. The algorithm uses a μ-calculus formula to describe the set of winning regions of the players.

In [14] the formula representing this winning set is described as a greatest

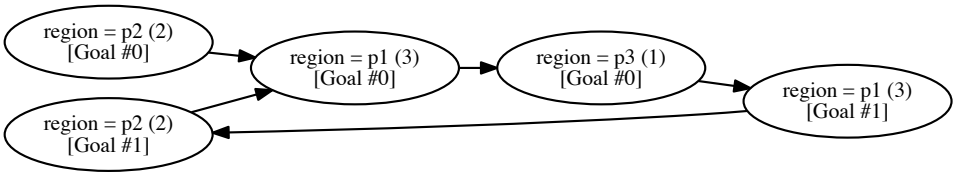


Figure 3.2.: Example automaton for a winning strategy

fixpoint that stands for the set of states in which the system is indefinitely violating the assumption of the implication or can get to a state in a finite number of steps, from which the system can win. In addition to that fixpoint there are some more constraints, like staying in states of which we do not know yet if they are causing the system to lose.

With the help of artefacts created by applying the algorithm to check realizability, it is able to produce an automaton that represents a winning strategy for the specification [3] and also an implementation in JTLV is given. For our simple example such an automaton looks like in Figure 3.2. The states consist of the current region in parentheses and the current goal it wants to reach in square brackets.

3.2. Linear Temporal Logic MissiOn Planning

There are several tools and libraries that are related to controller synthesis, such as slugs [6], LTLMoP [8], Demiurge[1], Pessoa [16] and AbsSynthe [5]. We have looked into these tools and based on the input and output language, documentation status and set of features, we have chosen LTLMoP as a basis of our implementation.

The *Linear Temporal Logic MissiOn Planning* (LTLMoP) [8] is a toolkit written in Python that allows the user to create a map and to write a specification for a robot in a known environment. To help the user with this task it offers a graphical user interface for creating the map and writing the specification. That specification can then be synthesized into an automaton, which represents the controller for the robot, using potentially different back ends, but at the time of writing only a framework for developing verification algorithms called JTLV [18] is supported. LTLMoP also allows the user to run a simulation with different robot configurations and simulation software by having a plugin like structure. Figure 3.3 shows the components of the toolkit and how they are connected. The grey components have a graphical interface, while the white ones are internal.

[1] `https://www.iaik.tugraz.at/content/research/opensource/demiurge/`

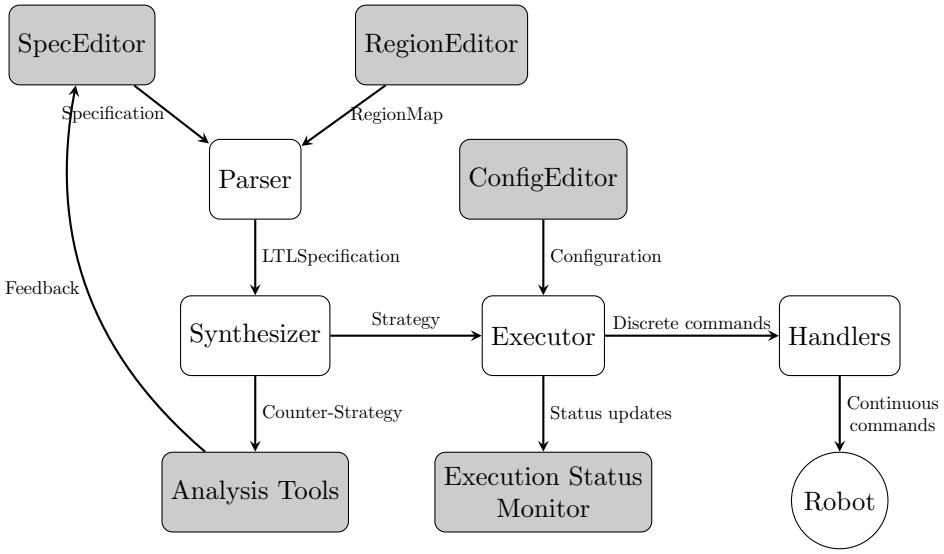


Figure 3.3.: Overview of the interaction and information flow of LTLMoP. Adapted from `https://github.com/VerifiableRobotics/LTLMoP/wiki/Overview`

First the specification has to be defined in either LTL directly or a format called *Structured English* which is a subset of English and defines keywords like *visit region* or *go to region* that get translated to $\square \lozenge$ *region*, where *region* is a defined region in the map. Then there is the keyword *always* with its LTL equivalent $\square \varphi$ which could be used for safety requirements, i.e. if the robot should never go to the kitchen. This language also has a way of describing implications and equivalence $c \implies r$ and $c \iff r$ with *if c then r* and *c if and only if r* respectively. Not only locations or regions can be described in the specification, but sensor or other custom propositions as well, to enable the user to react to certain events monitored by sensors or to have a memory of events happened. Because a robot typically is not only able to move but also to perform actions, like picking up an item or saying a sentence, there are action propositions as well. It is worth noting that all propositions are binary though, so a sensor input is rather below or above some threshold instead of some numerical value, and actions are either active or not. The full grammar specification can be found online [13].

From that specification LTLMoP generates an automaton of a winning strategy in case it is synthesizable or tells the user why it is unsynthesizable otherwise. If it is unsynthesizable it also allows to visualize a counter-strategy where the user can choose the actions step-wise [20].

Once the strategy is synthesized the simulation can be run with a specific configuration. The configuration consists of handlers for different aspects of the robot, like getting the pose, planning the movement from one region to another

or actually sending the motion primitives to the robot.

The most basic simulation is a dot that represents the robot which moves around in the very same map that was defined beforehand.

3.3. Robot Operating System

The Robot Operating System (ROS) [10] is a framework for writing software for robots. It has grown to have a variety of packages and tools ready to use for many robots. Some notable ones are simulation in 3D with the Gazebo [9] simulator as well as a functional localization and navigation stack.

When ROS is started a master node gets initialized. The master node is responsible for tracking which nodes are registered as well as setting up communication between nodes. Communication is done by subscribing and publishing to topics, to move the robot there is typically a topic several nodes can publish messages to and a node that subscribes to it and gives the movement commands to the actual hardware. Since there can be a lot of nodes and topics involved in a complex system the communication could be the bottleneck if everything had to be sent over the master node. Instead, the master only keeps track of the registered nodes and the related topics. If a node subscribes to a topic the direct address is given to the node, so that communication is done directly between nodes and does not involve the master. This architecture has the great advantage that modules can be swapped as long as they adhere to the interface given by the topic's message type. So if a sensor is replaced by another one the driver node simply has to publish the same message type and anything which used the old sensor will still work.

Since our goal was to use the hierarchical approach on either real Turtlebots or at least simulations of it, and ROS offers finished packages for those, it became our main target. The Turtlebot is a robot that consists of a mobile base with a differential drive, a 3D camera and a notebook that operates it.

We will look a bit into how localization of the robot is done when using ROS by describing the basics of the Monte Carlo Localization and how the local motion planning for collision avoidance that is based on the dynamic window approach.

3.3.1. Localization with AMCL

The navigation stack of ROS uses an adaptive Monte Carlo Localization (AMCL), which is based on the Monte Carlo localization [12]. There are two different kinds of localization described, *global localization* and *position tracking*. As the name suggests, global localization is the problem of finding out the position of

the robot in the map, whereas position tracking considers updating the robot's position when moving and sensing.

Initially, the robot's belief is uniformly distributed over the map if it doesn't know its initial position, and at the center of the position otherwise.

Then, the general idea of the localization is to update the belief of the robot's position by random samples, which consist of a robot pose and some weighting factor. For the position tracking there are two phases.

When the robot moves, the localization generates new samples that represent possible positions after the movement, based on previous samples. This means the samples get more diverse over time because of uncertainty of the robot's actual movement due to slipping or similar.

To regain some accuracy, the other phase is sensing, where the sensor input gets processed and the weights of the sample distribution gets recalculated.

Adaptiveness plays a role in the size of the sample sets. When doing position tracking, the sample sets can be relatively small and still provide useful positions, because the uncertainty is not as high. If the robot has no idea where it is, more samples have to be considered, so a bigger size of the sample set is chosen.

3.3.2. Local Motion Planning with DWA

The *Dynamic Window Approach* (DWA) [11] is primarily used to avoid collisions of the robot with unknown obstacles.

In order to move the robot and avoid the obstacles, the dynamics of the robot have to be known, so they can be used in the algorithm to only search for trajectories that are possible with respect to the dynamics.

As the number of possible trajectories is large, the velocities are reduced to smaller numbers in several steps.

First, only the trajectories resulting from curvatures are considered. These are induced by tuples of translational and rotational velocities.

The second step is to restrict these curvatures to those which are considered safe, meaning the robot has to be able to stop before it hits some obstacle.

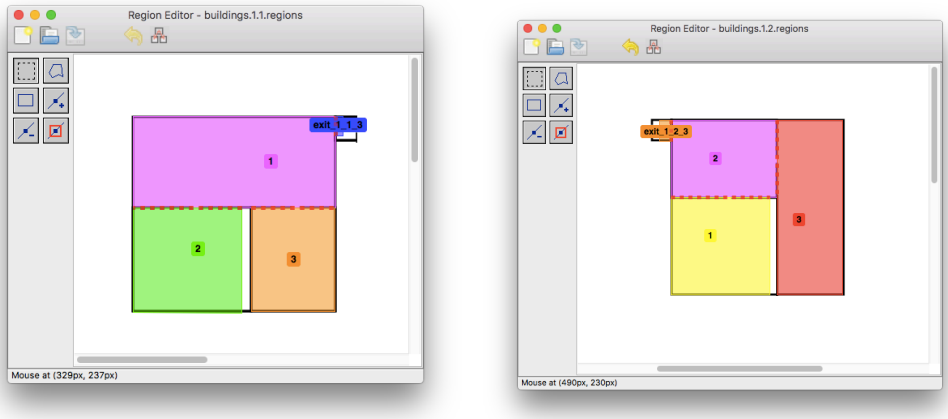
As a last step the remaining velocities are restricted to the dynamic window, meaning the only velocities which are kept, are the ones that can be reached by the robot's dynamics in a short amount of time.

The implementation of DWA in ROS then scores each simulated trajectory with respect to the proximity to the global path, goal, and obstacles and chooses the one based on that to move robot according to it.

4. Hierarchical Approach

Although our work is focused on the implementation of the algorithm, we will first describe the approach in [23] to introduce a hierarchy informally. That hierarchy splits a specification into different layers and thus smaller games, which can be solved faster, and chaining these together to satisfy a global specification.

The scenario has to be broken down into a hierarchy by the user manually. We will continue to use the example that we had outlined in Section 3.1. As a refinement for the example we have chosen the following: the buildings are the highest, rooms in buildings the intermediate and individual rooms on the lowest level. The intermediate level of the two buildings can be seen in Figure 4.1.



(a) Building 1, consisting of Room 1 (top), Room 2 (bottom left) and Room 3 (bottom right)

(b) Building 2, consisting of Room 2 (top left), Room 1 (bottom left) and Room 3 (right)

Figure 4.1.: Buildings example on the intermediate layer in the LTLMoP region editor

Every layer above the lowest one is considered abstract in the sense that these layers do not have to know the precise position of the robot at every time step and they do not move the robot directly. Instead, they pass their goal, if they have one, down to the next lower level. On higher levels, only abstract positions matter and thus less states are needed. On the lowest level the region of the game is restricted to a small subarea, which is why the individual games are

smaller.

Games are faster to solve this way, but it also implies communication between levels in a top-down manner. In our example: We want to move from building 1 to building 2, the highest layer in our abstraction would start a transition from building 1 to region 3, because we have to go through that to reach building 2, and pass that goal down to the intermediate level. This level would know the robot is in room 1 and has to simply go to the exit leading to the next building, and pass this goal down to the lowest level of games, which represent a room, like in Figure 4.2.

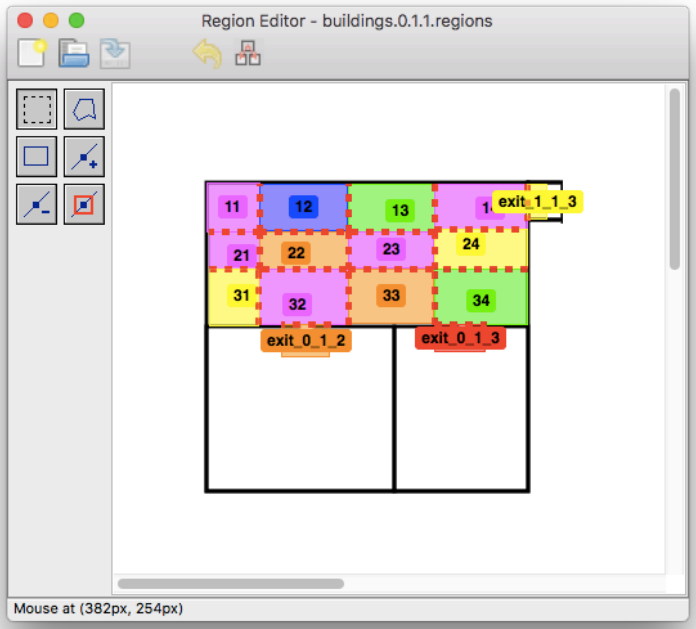


Figure 4.2.: Room 1 in building 1 in the LTLMoP region editor

Finally, to move the robot from room 1 to the exit, the game on the lowest level is started, with the `exit_1_1_3` leading to the next region as its goal and sends controls to the robot.

When a local game is finished, for example by reaching its destination in a reachability specification, it triggers a state change at the next higher level and this also means communication is done bottom-up as well. This communication between layers is repeated until either the global goal is reached or some game cannot be finished or instantiated, for example because the goal is blocked by the environment or the goal contradicts the local specification.

A proof that the theoretical algorithm for dynamic hierarchical reactive controller synthesis is sound can be found in [23]. Although the original hierarchical algorithm uses assume-admissible synthesis [4] as preferred way of finding winning strategies, we are not using these, mainly because the tools we have chosen do not support it at the moment. However, the proof of soundness of the hierarchical games does not rely on this special class of strategies, as outlined in [23], so we can choose GR(1) as subclass of LTL for our specifications.

Now that we have explained the general idea of the hierarchical approach, we dive into the way we have implemented it and how we have mapped the theoretical part to the implementation.

4.1. Implementation with LTLMoP

The theoretical work is not translated one-to-one into the implementation. Instead, some things are handled differently. For one, the paper describes the system states as a grid where every cell is just as big as the robot itself, so it can be in exactly one cell at a time. When modeling this in a tool, it is very cumbersome and to some extent not necessary. We allow bigger regions and let the controller move between them. This allows us to build scenarios faster, because every region has to be created manually, but we could also be as precise and make regions as big as the robot and thus have a grid as well.

Furthermore, obstacles are different too. Obstacles are treated as either static, i.e. walls inside of a room, sensed and thus indicated by sensor inputs or dynamic, which are not mentioned in the specification at all. In [23], the obstacles are represented as a set of grid cells, that can either be occupied or not at a specific point in time and can change according to the definition of the environment transitions.

Those environment transitions cannot be specified in the implementation, only assumptions can be stated. System transitions are another part being treated in another way. Instead of describing which state the system can transition to, the map of regions includes information about adjacent regions, so the system transitions are given implicitly, since the robot can move to every adjacent region, and do not have to be dealt with manually.

As we focused on scenarios in buildings and our target robots typically can not open doors, they are an important factor and should be handled in some way or another. One approach to tackle them is to define the state doors as sensor input, so something like *if the door1 is not open, we cannot go there* can be written, together with the assumption, that the door is infinitely often open. This would imply the need to have some way to sense if a door is open or not, like a sensor inside of the door that is connected to the robot over network or

some central station the robot could ask. The benefit of this is that it is cleanly integrated into the specification and can be reacted to, but it also assumes some infrastructure for sensing it.

Another way is to treat doors as any other dynamic obstacle, which means that if the robot recognizes it can not go to its goal, it has to either stop or to find another way. So when setting the goal to *go to door1 or door2*, in the case that *door1* is blocked the goal *door2* is still available.

The first approach is fine for the sole purpose of simulation or some very specific testing environment, where these kinds of sensors can be deployed, but for the majority of scenarios it would not be applicable. Therefore we decided to treat doors the same as dynamic obstacles.

As far as the implementation is concerned, several parts are needed: the specification, something that represents the environment and its regions, game solving, and the interaction between layers. Because the specification and definition of map or regions depends on the implementation of the game solving or controller synthesis, we first looked at existing solutions for that.

In the end, we decided to use LTLMoP, because it had a graphical interface for the most steps needed and its architecture seemed to enable us to leverage the existing code and add our implementation to it. Also, compared to other tools, it allows to write the specification in *Structured English* as well as LTL, which is a nice addition.

4.1.1. Overview

To implement the algorithm mentioned we treat every local game as an instance of LTLMoP, with its own defined regions, specification and configuration. On top of that a component we have called `LocalGame` is used. The purpose of this component is to handle a game, as the name suggests, which means it should be able to change the goal of a game, synthesize the specification and communicate with its parent or child layer if existent.

Since LTLMoP uses a remote procedure call (RPC) library to communicate between processes, namely the executor and everything connected to it, and the graphical user interface for the simulation, we decided to use that pattern for our communication as well. This means the `LocalGame` needs to know the port of the parent. Because the `LocalGame` is started by our implementation of an `AbstractHandler`, we can pass this port to the constructor of the game.

To give an overview how the components are connected and which instances exist in our example, Figure 4.3 shows the instances of the main components and the communication directions of the individual components. Every row in the figure represents one game, starting with the `LocalGame` that runs the game and therefore starting the `Executor`, which in turn uses various handlers, with

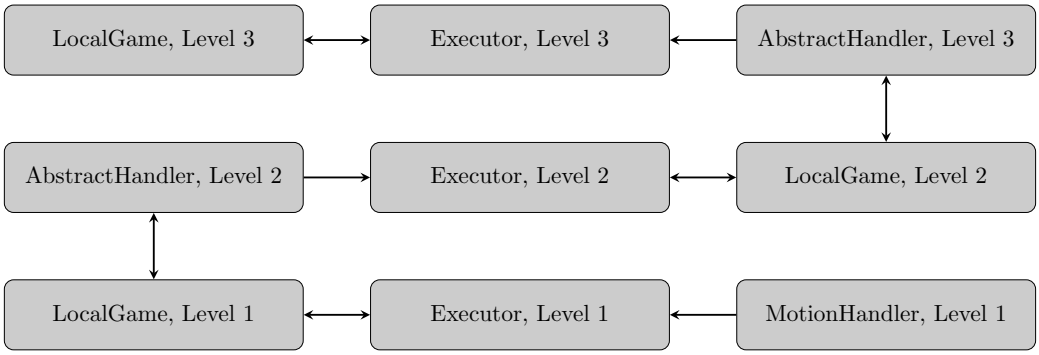


Figure 4.3.: Communication between components in the three layer buildings example

the `AbstractHandler` being the most important one for games of higher levels.

Also notice the unidirectional arrow between the handlers and the executor. This is due to the fact that the executor should not have any knowledge about the handlers, and for compatibility reasons and because the needed communication is a special case we did not include it in the interface the handlers should implement.

Furthermore, note that the executor has many handlers, not only the two types we have shown here. The typical handlers an executor has are one of each shown in Figure 4.4. All of these handlers have a reference to the instance of the executor they belong to, so they can interact with the graphical user interface as well as other handlers, if they need to. The `MotionControlHandler` is used for the region to region movement, which can include sending the low-level commands to the robot. If not, it can rely on the `DriveHandler` and `LocomotionCommandHandler`. As an example, differential-drive robots cannot use a two-dimensional velocity command directly, instead they have to use a velocity and some rotation to achieve the same movement, which is the responsibility of the `DriveHandler`. The `LocomotionCommandHandler` then just has to send the correct command depending on the robot type. All the other handlers are self-explanatory.

4.1.2. AbstractHandler

As seen in Figure 4.4, our implementation of the `AbstractHandler` is done as a subclass of the `MotionControlHandler`. Instead of moving the robot from one region directly or through the drive- and locomotion-handlers, we instantiate a game of the next lower level.

In Algorithm 1 the pseudo code of the AbstractHandler is shown. In addition to that procedure, an RPC server is set up to listen to events from the local game, since the handler is the only connection between the current game and the newly

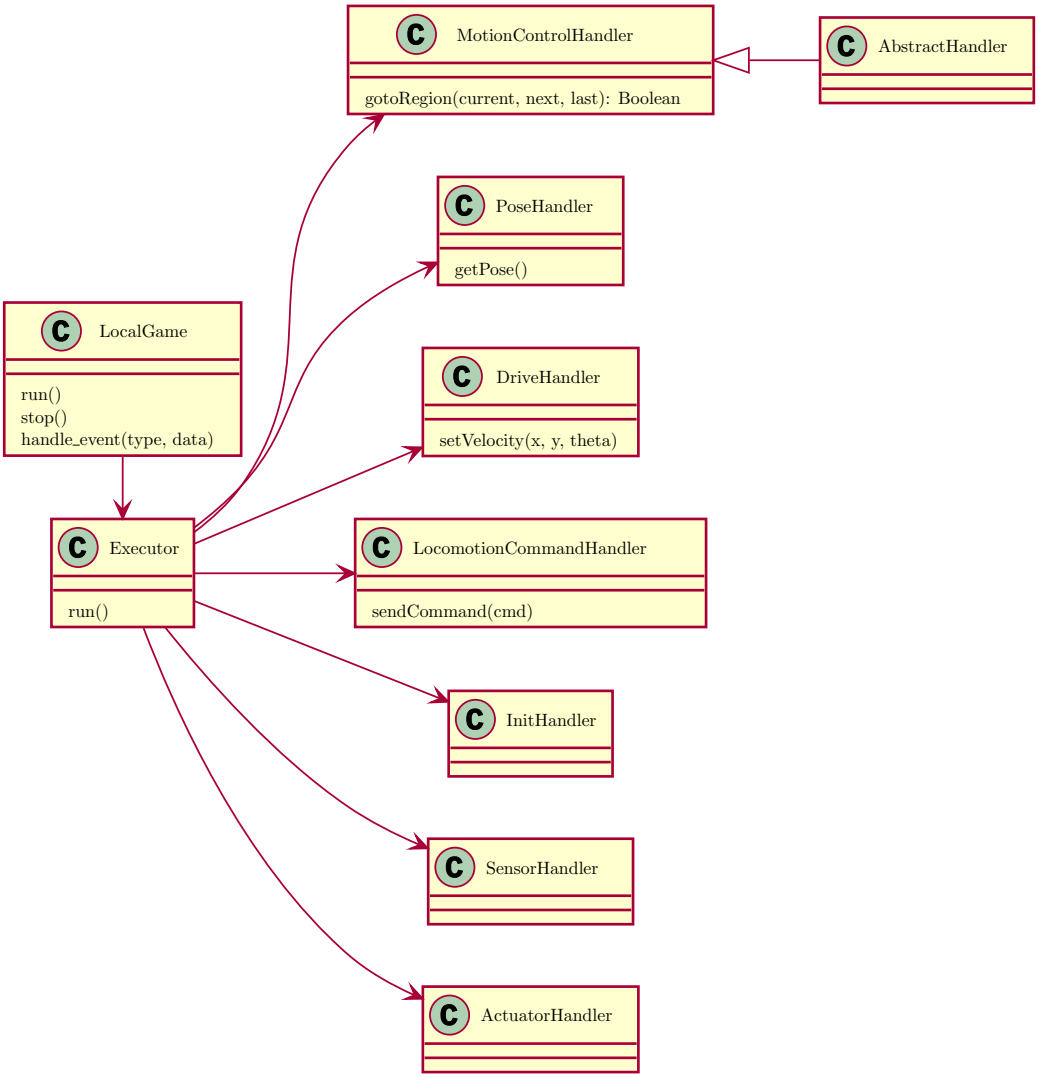


Figure 4.4.: Typical Handlers of LTLMoP used by the executor

created one. If the event is not meant for this handler or game, it is just passed on in the hierarchy until it reaches the correct component. But there are events targeted at the `AbstractHandler`, such as crossing borders or if the local game reports it is not synthesizable.

Algorithm 1 AbstractHandler

```
1:  procedure gotoRegion(current_region, next_region)
2:      if arrived is True then
3:          stop local game
4:          return True
5:      check for goal
6:      if a game is running and next_region != current_goal then
7:          stop local game
8:      if there is no game running then
9:          set arrived to False
10:         create a local game
11:         start the local game
12:     return arrived
```

In line 5 we check for the goal, because some additional steps are needed when deriving the goal from the next region. As such, it can happen that the next region is the same region the robot is in. In that case, we set the goal to `None`, which allows us to stay in the current region on that level and execute the specifications of it. As an example, assume the robot is in the cafeteria and should stay there to take orders. The region should not be changed, as the robot should stay in the cafeteria, but the specification, that represents the handling of orders, should still be executed. Even if the special case of no goal is not applicable, the internal representation of the region has to be translated to the actual name used in the map, which is also done in that line.

As a next step, we check if there already is a game with a different goal running. This could be related to a sensor change in the upper level for example. To return to the cafeteria example, a sensor could check the time, and when the robot's shift is over, it could change the goal to the room the robot's battery is charged in. The local game would be stopped, so the check if a game is running shortly later is evaluating to true and a game with the new goal can be started.

The procedure shown above does not actually handle the state transitions directly, this is done by the `handle_event` method, which is part of the `AbstractHandler`'s RPC interface. Once the border is crossed to the goal region in the local game `handle_event` is called by the local game with the type `BORDER` and thus the instance variable `arrived` is set to `True`. After it is set to `True`, the next iteration of `gotoRegion` will return `True` to its executor as well, triggering a state change and

the current and next regions with it.

To actually create a game with goal in the `AbstractHandler`, the current region as well as the goal get passed to the constructor of `LocalGame`, which we will describe in the following.

4.1.3. LocalGame

The local game is the starting point of the algorithm, so to start a scenario the first thing that will be created is a `LocalGame` of the highest level with no goal, which starts the execution of the game of that level. As mentioned before, this will also set up an RPC server for the `LocalGame` and pass the port to the executor. That way, they can communicate in both directions.

On initialization of a `LocalGame`, several things are done: some information about the game are extracted from the file name and the specification. We assume the file name is of some specific form that contains the basic name of the project, the hierarchical level and the path to the game in the hierarchical sense, joined by periods. If we take our example of the buildings, the local game on level 0 in building 1, room 2 would be *buildings.0.1.2*. As the specification needs a region file as well, the original files have to be named *buildings.0.1.2.spec* and *buildings.0.1.2.regions*.

While we do not have to change the regions, the specification might have to be changed, for example if we want to set another goal to that game. This is because we need a specification for the local game, applied to our buildings example this could be the specification to visit all rooms inside of a building if we have no goal set by the higher level and therefore do not need to go to another building. But we still want to visit all rooms of the building and then move to the next building in case the higher level dictates it, so we have to change the specification.

Thus, we need to read in the specification file and to be flexible we keep it as a list of lines. To set a goal we can then append the line `go to exits`, where `exits` are the regions leading to the goal, without touching the rest of the specification. There are several ways to acquire the exits. If it is given by the parent and starts with "exit", we keep that name as goal. In case it is another name, we have to read the region file and check for the correct pattern. As a convention, we assume exits have the special pattern `exit_level_from_to`, where `exit` is the literal string "exit", `level` is the hierarchical level this exit represents, and `from` and `to` are the regions the exit is connecting on that level. Including the level in the name is used so we do not need regions that represent exits to exits and to distinguish between regions of one level from regions of another level. In our example it can be seen in room 1, where one region is called `exit_0_1_3` and another one `exit_1_1_3`. The first one is the connection to the room 3 on the

same level, the latter is connecting building 1 to building 3.

After that, we build a new file path for the specification by building the SHA1 hash value of the list of specification lines, appending it to the file path and writing the updated specification to it. This allows us to keep already synthesized strategies for known specifications, saving us from redoing it every time. Note however, that we still have to synthesize a strategy possibly several times per region, because the specification is different for every goal region the level above has.

After the initialization of the `LocalGame` is done, it can be started by calling the `run` method. The loop and main idea is described in Algorithm 2. The variable `had_changes` is a variable that indicates whether there were any changes made to the specification, thus meaning we have to check if we need to synthesize it again. Reasons for this can appear if something unforeseen has happened while executing the game and we need to adapt. The details for this will be discussed in Section 5.2, since these should not happen in a basic simulation of the games. Checking if the current specification needs to be synthesized is done by `is_dirty()`, which checks if the file appended with the hash value of the specification is already existing or not.

In case it is not existing, we need to synthesize a strategy. To do so the `SpecCompiler` class provided by LTLMoP is instantiated with the current specification path and then compiled. This may fail, because the specification is actually unrealizable, in which case the parent is sent a message of type `UNSYNTH` with the goal region as data. `synthesize` itself returns either `True` if was unrealizable or `False` otherwise.

Algorithm 2 LocalGame

```
procedure RUN()
    set had_changes
    while had_changes do
        clear had_changes
        if is_dirty() then                          ▷ Check if file is existent
            if !synthesize() then
                break                               ▷ Unrealizable → stop game
        set up executor
        wait until game is done
        store outputs of the game
        stop executor
```

On the other hand, if the `LocalGame` is receiving an `UNSYNTH` message, it means the lower game could not be synthesized and therefore we have to change something. Since we look at an specification of type GR(1) and the order is arbitrary,

reordering the places to visit can resolve the issue, because some temporary constraint to not go into a region, which is necessary on the path to the goal, might be gone later. Thus, we search for the line `visit region`, where `region` is that goal in our specification list and and we try to move it to the very end. Should the goal be on the last line already, we stop the game, because we know no other means to resolve the conflict. But if that is not the case we write this specification to the file, after updating the path with the new hash value. If the new specification is synthesizable, we proceed to run it, by setting the `had_changes` flag so the next iteration of the run loop will be run.

4.1.4. Memory propositions

As we have seen in the previous section, the execution of a local game might be restarted. If we just restart the game with a changed specification, it would try to reach the first goal again. To be more specific, consider the example seen in Figure4.5a.

The robot starts in region 1 and then tries to region 2. However, if we forbid the region which represents the exit to region 2 in the local specification for demonstrating purposes, the specification will not be able to be synthesized. The resolution will then try to reorder the regions and restart the game. This would imply the robot again tries to reach region 2 first. Since we already have visited it, we do not want to go there again, so we need a way to express that.

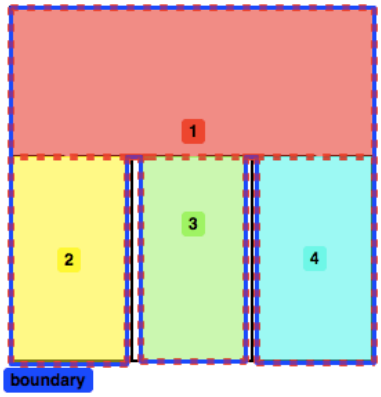
As a solution to this we have considered either storing the visited regions somewhere and hooking into the running automaton to change to another state that is trying to reach the next goal, or to add information about that to the specification. As LTLMoP supports having custom propositions and these can be used as such, that solution seems to be the cleaner one.

In 4.5b a specification that uses these propositions is shown. Propositions that represent the memory are added and guards are placed around the temporary goals. The specification now tries to visit the regions only, if the proposition is not true already. In this example, the propositions are set when the corresponding region is visited, but never reset, which means the robot would stop after visiting all regions, because no goal is left to reach.

Those propositions can not only be set and reset on regions, but any Boolean formula. For the purpose of memorizing where the robot was already, taking only regions into account is enough.

4.1.5. Changes to the Existing Code

As discussed in the previous sections, detecting the crossing of a region border is important for our implementation. But to get this information we had to mod-



(a) Example of a scenario with memory propositions

```
if not v1 then visit 1
if not v2 then visit 2
if not v3 then visit 3
if not v4 then visit 4

v1 is set on 1 and reset on false
v2 is set on 2 and reset on false
v3 is set on 3 and reset on false
v4 is set on 4 and reset on false
```

(b) Specification with memory propositions

ify the existing code of the `LTLMoPExecutor` and the `ExecutorStrategyExtensions`, which handle execution of the automaton that represents the winning strategy. To be precise, we have added a new target for remote procedure calls, that gets set when a `LocalGame` instantiates an `LTLMoPExecutor`. Instead of sending messages to the graphical interface we are now able to send messages or call methods of the `LocalGame` corresponding to this executor. Sending messages is implemented by calling the method `handle_event` on the remote target, with a string as message type and an arbitrary object as message data. This way, we can send a message to the `LocalGame` when the executor detects a border change, by sending a message of type `BORDER` with the name of the region we went into as data, to the `handle_event` method of the `LocalGame`. If it is a game of lower level and the region is an exit to another game, this region is passed to the `AbstractHandler` above, to decide what the next step should be.

Furthermore we have exposed the output propositions of an executor as a method, so we can get the current state when we stop a game and set it as initial truths of the next game, if we want to. This will prove beneficial when we want to change the specification of the current game and restart it with the new specification. The initial truths can be set through the configuration as well, so we had to merge those with the ones we get when initializing, in that case we simply override the initial values by the newer ones.

When running an LTLMoP simulation, the user had to press a button to start the execution. Because we are starting multiple simulations by using the hierarchical approach starting every game is not desirable, so we have changed it to start the execution when it is run.

Apart from these changes we have also added some methods for convenience. Internally LTLMoP uses mostly subregions that get created when synthesizing

the specification. Since the specification itself has different names and we want to use those, we have added methods to find the original region name by the subregion name.

As a last change we modified the name of the counter-strategy, which the GR(1) implementation creates if the specification was not realizable. It was the same name that gets used for a valid strategy, and since we check for existence of this file we had to change it, so it does not get mistaken for a valid strategy.

4.1.6. Specifics to the Basic Simulation

While the previously described changes and additions could be used in other simulations as well, we had to make some additional changes to get a more realistic result with the basic simulation of LTLMoP.

One issue is, that the position handler of the basic simulation needs an initial position. Where in a more realistic setting the robot would provide a pose, here the initial pose has to be given in the form of a region. Since we need a generic way of telling the simulator in which region we start, we have added some variables that keep track of the last region the robot was in in the `AbstractHandler`, so when the goal changes there, the new game can be given its last region as initial one. The higher levels have to know the region of the lower levels, which in principle breaks the idea of hierarchies, but it is only used for the simulation and would not be used in the more realistic version. Also, the algorithm itself does not depend on it.

But also creating new local games when exits are reached need more information, because it is not know which region in the next game the exit corresponds to. To overcome that lack of information we are checking if a file named *mappings.json* exists in the same directory the specifications are in, and use those information to set the correct region in the new game based on the mapping defined in that file. This file should contain of a nested *JavaScript Object Notation* (JSON) object, that represents the regions in the topmost nesting depth, the hierarchical layer on the next depth, and finally the mapping of exit to region it corresponds to as key and value respectively.

Again, the `AbstractHandler` uses these information to set the last known region to the one from the mapping before it starts the new game, so the simulation of the next game starts in the correct region.

4.2. Evaluation

One important goal of the hierarchical approach was to see if the reduced complexity of the games result in an improvement in terms of time needed to synthesize or size of possible games.

In order to see how the hierarchical approach performs in comparison to the monolithic one, we have used the example of two buildings connected by some other area from the beginning of the chapter again. The task of the robot is to simply patrol between every room in every building. The layout and regions of the monolithic version can be seen in Figure 4.6.

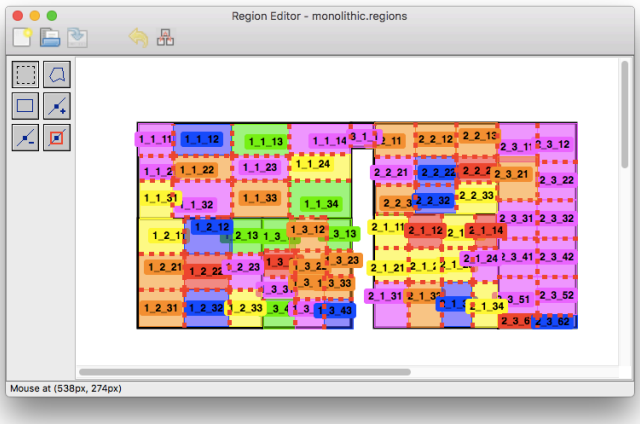


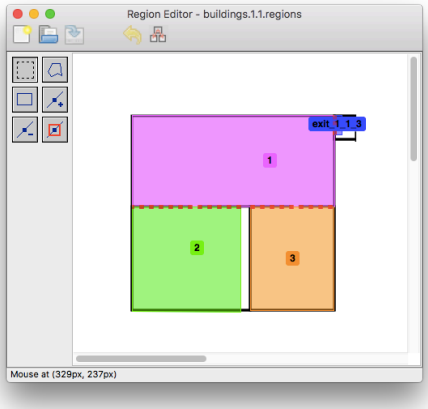
Figure 4.6.: Monolithic version of the buildings example

The corresponding hierarchical game is the same as in the introduction of this chapter, but the figures are shown again for convenience. Figure 4.7 represents the buildings, and the exemplary room 1 of building 1 in Figure 4.8.

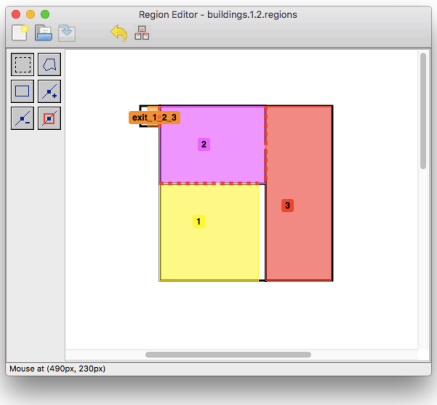
Preparing the scenario in a hierarchical manner takes a lot of time as it currently is, because not only the way how it can be divided into layers has to be determined, but also because the individual layers and games have to be set up. Every game has to have its own defined regions of the corresponding subarea of the original scenario and its own specification. While the graphical interface of LTLMoP to edit regions tends to be unresponsive for a large number of regions (like our monolithic example of the buildings), it is still faster to set up only this one big game instead of creating 11 individual games. This set up time could be improved by adapting the tooling to allow an efficient way to edit multiple games, but this has not been implemented yet.

4.2.1. Live lock because of reordering

A problem that exists in reactive controller synthesis could not be solved: if the robot has to choose between two doors, depending if one or the other is open, it might come to a live lock. Assume it chooses the first door, because that one is open. The robot will start to move towards this goal, but before it can reach



(a) Building 1, consisting of Room 1 (top), Room 2 (bottom left) and Room 3 (bottom right)



(b) Building 2, consisting of Room 2 (top left), Room 1 (bottom left) and Room 3 (right)

Figure 4.7.: Buildings example on the intermediate layer in the LTLMoP region editor

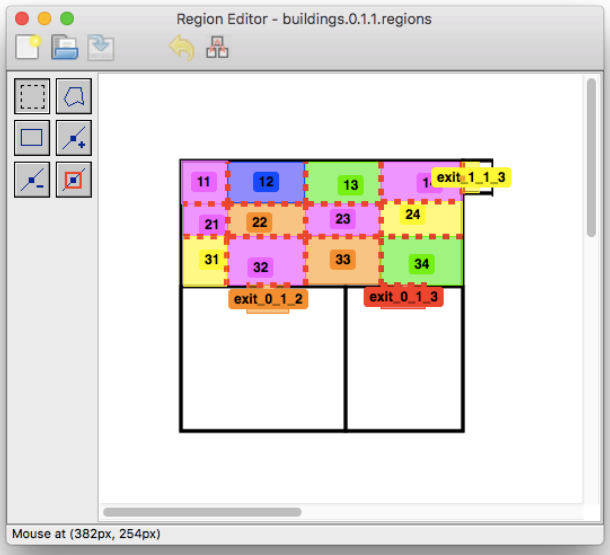


Figure 4.8.: Room 1 in building 1 in the LTLMoP region editor

it the door is closed. Since the second door is open, it will pursue this new goal. Again, before it can reach it the door is closed and the first one is opened again. This can potentially continue forever, creating a live lock where the robot satisfies the specification, but will never reach its goal.

To demonstrate this, we have built a hierarchical scenario with two levels, where the robot starts in the middle of three regions and the specification states it has to go to the left and right region. The lowest level specification for the center region however forbids to go to the exits to left and right. The two specifications are shown in Figure 4.9, the complete example is in B.2.

```
visit left
visit right
```

(a) Specification of the top level

```
always not exit_0_mid_left
always not exit_0_mid_right
```

(b) Specification of the center game on level 0

Figure 4.9.: Specifications of the live lock example

Once the game is started, it will try to synthesize the low level game of the center region with either goal. Because the specification is unsatisfiable in both cases, it will fail and propagate that information to the higher level. Our resolution then tries to reorder the regions and chooses the other side as the next goal, which will fail again, continuing the alternation of sides forever.

One workaround for this particular case could be to count the number of consecutive failures and abort after some threshold. While this would solve the problem in that static example, mimicking the behaviour of the monolithic approach which would return that the specification is not realizable, it would not help where the change of goals is induced by doors or sensor input, so we decided not to implement such a failure counting.

4.2.2. Resource usage

At first we thought the time needed to synthesize one monolithical game would take much longer than several small games, because the complexity is still cubic. Benchmarks supported this, until we have noticed, that LTLMoP is spending most of the time to process the regions of the map. To be more precise, it checks for overlapping regions to create non-overlapping regions and decomposing. If region A and B are overlapping, it creates the subregions $A \setminus B$, $B \setminus A$ and $A \cap B$. Decomposing is needed if so called non-projective locative propositions are used in the specification. These could be `near`, `between` or `within`. In that case again subregions have to be created to represent those regions.

Since every region is compared with every other region and checking for over-

laps seems to be computationally complex enough to have a big impact, this makes up for most of the time needed to compute a strategy.

Because of this additional time we decided to turn off the decomposition of regions and shift the weight onto the user, who has to make sure regions do not overlap. It also means these locative propositions can not be used and also obstacles are not supported.

The times of the time it took to synthesize the strategy on our used computer can be seen in Table 4.1. The tests were run on an Intel i5-2500 with 8GB of RAM on Ubuntu 16.04. Because of the issues with decomposition mentioned before, we stopped the monolithic version after over 60 minutes, although execution time was only about 758s at that time, according to the timing tool. To time the execution time the tool *perf stat* was used to run the compilation 10 times, the average time used can be seen in the table. To calculate the sum of the individual hierarchical games we have taken into account that games have to be synthesized more than once, because of the different goals. The typical sequence of synthesizing the games for this scenario is the following: $2^2, 1^1, 1_1, 2_1, 1_1, 3_1, 1_1, 3^1, 1_3, 2^1, 2_2, 1_2, 2_2, 3_2, 2_2, 1_3$, where the superscript is the level if different from the lowest and the subscript being the region of the next higher level. After those, every possible region and goal combination needed is synthesized and can be reused. For the sum we have multiplied the number of occurrences of every game with the time used to synthesize it with one goal only, we did not change goals, because for the time to synthesize the difference in goals is insignificant.

Game	**# Regions**	**Time in** s	**Time in** s (no decomposition)
Buildings(Monolithic)	67	>758	1.20
Buildings(Sum)	150	23.04	6.25
Buildings.2	3	0.34	0.35
Buildings.1.1	4	0.35	0.39
Buildings.1.2	4	0.37	0.36
Buildings.1.3	3	0.34	0.34
Buildings.0.1.1	15	5.09	0.36
Buildings.0.1.2	10	0.59	0.43
Buildings.0.1.3	13	1.46	0.47
Buildings.0.3.1	3	0.35	0.34
Buildings.0.2.1	13	1.28	0.34
Buildings.0.2.2	12	0.58	0.45
Buildings.0.2.3	13	0.94	0.46

Table 4.1.: Benchmark of a monolithic versus the hierarchical version

Following these results, the hierarchical approach seems to take longer than the monolithical version if decomposition can be turned off. However, time is not the only limiting factor.

The amount of propositions that are used for memory blow up the state space, which can lead to the implementation running out of memory before finding a solution, especially if the memory of the system is limited. By splitting the games into smaller ones it is possible that propositions are only needed in some of the games instead of all of them and thus needing less memory.

Even if memory usage is the main concern, additional propositions increase the time needed to find a strategy as well. Some benchmarks for the monolithical against a hierarchical game can be seen in Table 4.2 with the respective specifications in Listing 4.10. The main reason why the hierarchical example is faster than the monolithic version is the number of regions. The total time of the hierarchical game would heavily rely on the kind of propositions needed.

```
                                  if not sensor1 then visit 1_3_12
if not sensor1 then visit 11      if not sensor2 then visit 1_2_11
if not sensor2 then visit 12      if not sensor3 then visit 1_3_11
if not sensor3 then visit 13
if not sensor4 then visit 23      if not sensor4 then visit 3_1_1
if not sensor5 then visit 33
if not sensor6 then visit 32      if not sensor5 then visit 2_2_11
                                  if not sensor6 then visit 2_3_11
go to exit_0_1_2                  visit 2_1_11
```

(a) Room 1 in building 1 (b) Monolithic version

Figure 4.10.: Example specifications with sensor propositions

In case the propositions are global, meaning that every game has to rely on all the sensor inputs, the total amount of time needed to compile every game could be even higher than in the monolithic case.

If the propositions are local to some games however, the time savings could be huge. To make this distinction between global and local propositions clear, consider these examples: A global proposition is one, that would be needed in every game in the hierarchical approach, like the robot should move to some region if the battery is running too low. It has to react in every possible game, thus is considered global. On the other hand, a local specification could be confined to a specific room. Assuming room 1 in building 1 is a cafeteria, the robot should not get too close to the buffet between 11:00 and 14:00 o' clock. This proposition, modeled as sensor input, is only needed in this one game instead of globally, so it is considered as local.

Local propositions do not affect the time needed to synthesize other games, so having mostly local propositions would decrease the overall time significantly.

Game	**# Regions**	**# Propositions**	**Time in** s
Monolithic	67	4	5.42
Monolithic	67	5	37.28
Monolithic	67	6	366.81
Buildings.0.1.1	15	4	1.01
Buildings.0.1.1	15	5	2.57
Buildings.0.1.1	15	6	12.44

Table 4.2.: Benchmark of some games with additional propositions

Independent of the memory and time issues the hierarchical approach has one big advantage: it allows to change specifications for lower levels while the system is running. Consider an environment that changes, because cubicles are moved. Once the robot is out of that particular room, the specification could be changed and the corresponding strategy file deleted. The next time the robot enters that room it will not find the strategy and therefore start to synthesize it without having to be restarted. This also means that only a part of the games has to be synthesized again, instead of the whole game in the monolithical approach.

5. Connecting it with the Real World

Having such a simple simulation is a good starting point, but we aimed to run it on a real robot or at least some more realistic simulation of it. Therefore we have searched for an integration of ROS in LTLMoP. There was a proof of concept this did that integration mentioned in [24], but their solution had some problems with our goal.

In their example, a world in Gazebo is generated on every run, based on the map in LTLMoP. It is a plane with the colors of the regions and obstacles are boxes of a fixed height. The position is obtained by asking Gazebo directly, so it is always accurate and thus the driving could be done by direct velocity commands in their `LocomotionCommandHandler` for ROS.

Since we wanted a more realistic simulation, we could not use their solution directly. First, we had to build the world for Gazebo, in which the robot should move. Then, the mapping between the LTLMoP map and the Gazebo world had to be established. And last, we had to get the position of the robot based on its navigation stack and move it using the same.

5.1. Connecting LTLMoP with ROS

Multiple steps are necessary to connect LTLMoP with ROS and Gazebo. First, we have created a world in Gazebo representing the scenario of the buildings. Secondly, we have written a launch file to start all the needed ROS nodes to run the example and lastly, the handlers that connect LTLMoP with ROS have been implemented.

5.1.1. Gazebo world

The world we created was an empty plane at first. Then we created a model of the building using Gazebo's building editor, which allows the user to import a floor plan and add walls, stairs and doors relatively easy. The resulting building complex can be partly seen in Figure 5.1, where also some cylindrical obstacles have been added. We have left the building as basic as possible and added obstacles only on demand, a more realistic approach could be a completely furnished office building or similar.

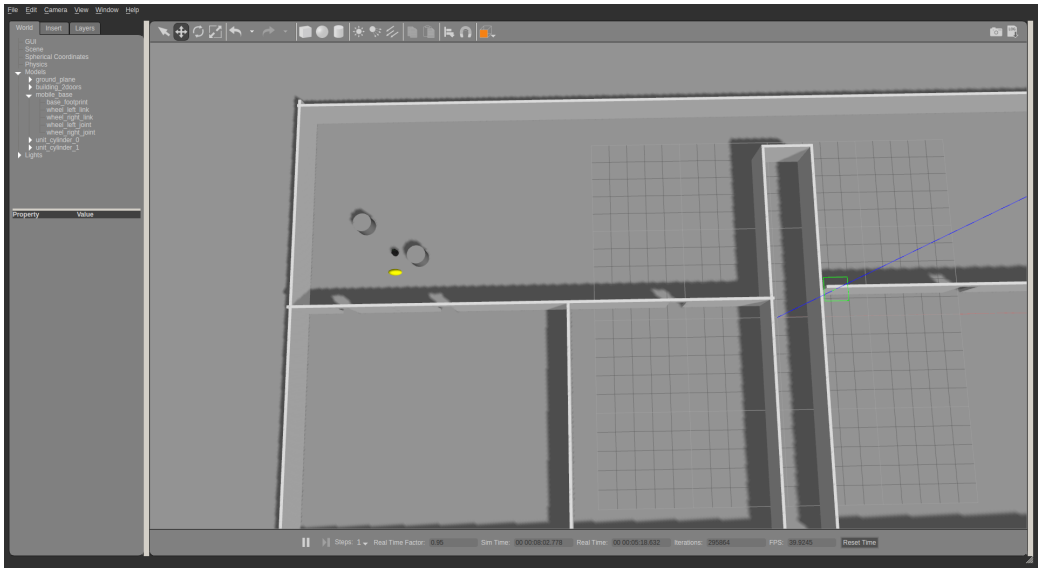


Figure 5.1.: Buildings example in Gazebo simulator

5.1.2. Launch file for ROS

This world file has to be included in the launch file, which actually starts the ROS nodes needed for execution. The file can be split into different parts: the Gazebo simulator loading the world, nodes involving the Turtlebot, such as configuration and spawning, and the navigation stack.

To start the Gazebo simulator it suffices to include the launch file of an empty world with a different world as parameter. Optionally, the flag if debug output should be turned on and if simulation time should be used can be set. Using simulation time is important, because it tells ROS that somebody else, in this case Gazebo, is publishing to the */clock* topic, which is used to check if messages are recent enough and because the simulation could be slower or faster than the real time, Gazebo should be the one to publish it.

Starting Gazebo alone will not spawn a robot, so this has to be done by another node. The ROS package *turtlebot_gazebo* for Turtlebot-Gazebo integration already includes a launch file to spawn a robot, so this is used with some default settings for sensors and the base.

The bigger portion of the launch file is the part concerning the navigation stack. There, a map server is used, that publishes the static map to a topic, which the localization and movement nodes will read from. These nodes are the *AMCL* node for localization and *move_base* for moving respectively. As the AMCL node operates on 2D laser scans rather than the 3D point cloud the camera is providing, an additional node that converts the point cloud to a laser scan is started as well.

Once the launch file is started, another tool called *Rviz* can be started. It can be used to visualize the perceived environment of the robot, or to visualize the current perception of the navigation node, including it's approximated pose with uncertainty and the planned path. Its interface for the navigation visualization can be seen in Figure 5.2. The bold lines are representing the walls provided by the static map and the two round shapes represent the cylindrical obstacles perceived with the laser scan as a local cost map. The lines from the black dot representing the robot are the simulated trajectories the robot considers as viable and the thin line on the top left of the upper obstacle is the global plan the robot is following right now. On top of that, Rviz also lets you send manual navigation commands and reset the perceived pose of the robot, which is useful if it gets stuck.

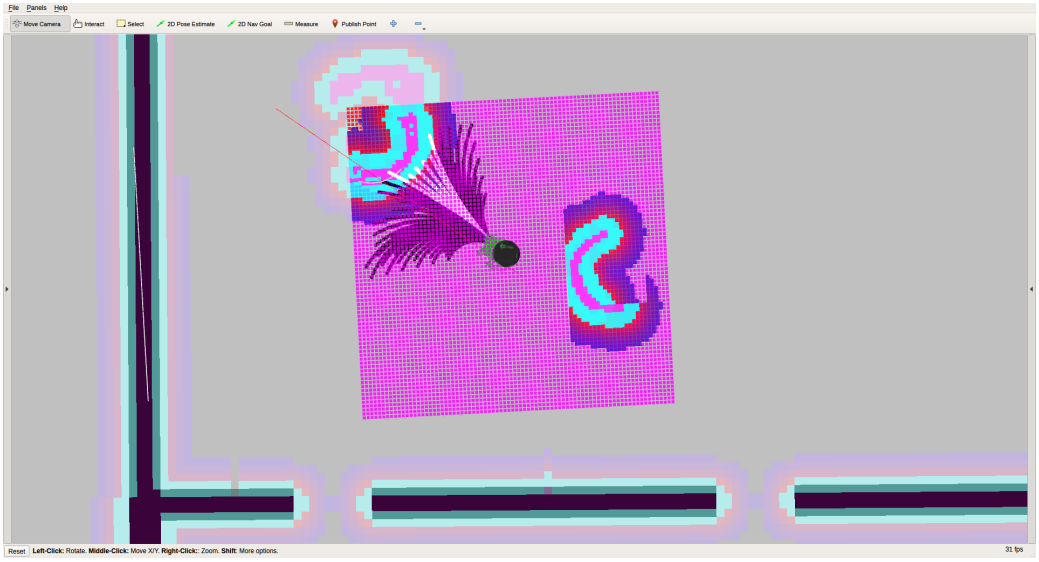


Figure 5.2.: Visualization of the navigation in Rviz

5.1.3. Handlers in LTLMoP

As a last step LTLMoP and ROS have to be connected. Again, LTLMoP's architecture with different handlers are of advantage. The default `PoseHandler` is replaced by a `RosHierarchicalPoseHandler`. It starts up another ROS node that listens to the transformation between the base link, which represents the robot, and the map, thus resulting in the position of the robot relative to the map. This only returns the approximated position of the robot though. In order to get the position as well as the covariance matrix, the pose handler would have to subscribe to the topic published by the AMCL node directly, but since we did not want to use that information we chose not to. The third way of getting the

position of the robot would be to ask Gazebo directly, as we mentioned before, which was the way the authors of LTLMoP chose.

Because the coordinate systems of LTLMoP and the Gazebo world can be different, a way to transform coordinates from one into another has to be provided. LTLMoP handles this already, by giving the user a calibration tool. To use it the user has to mark at least three points which should be checked in the LTLMoP map. After that one point after another are shown and the robot should be moved to that position manually. When it arrived a button can be pressed to get the current position by the pose handler. When the position of all points is known a calibration matrix is calculated and can be used in handlers to convert coordinates from one system into another through the functions `coordmap_map2lab` and `coordmap_lab2map`.

Using these functions, the `MoveBaseMotionHandler` is implemented. Again, it is a ROS node. As an implementation of the motion handler interface, the method `gotoRegion(current_region, next_region)` has to be satisfied. The general idea is very similar to the `AbstractHandler` mentioned in Section 4.1.2, but instead of creating another game, the coordinates of the goal, which is some point in the next region, are transformed to the world coordinates and sent to a `SimpleActionServer` of `move_base`, which then tries to reach that goal, by using a global and a local planner.

The complete algorithm can be seen in Section 5.2, where we discuss some additional problems and embedded some solutions into the handler.

In contrast to simple messages published or subscribed, these *ActionServers* and *ActionClients* are meant for tasks that should be tracked in the application. This means the status of the current goal can be asked for, which is useful to determine if the robot has arrived at the goal already or even more importantly to check whether the goal could not be reached.

Additional handlers or templates are provided by the LTLMoP authors, such as exemplary actuator handlers for a different type of robot called PR2, which in contrary to the Turtlebot, has arms that can be used. Templates for two types of sensor handler implementations had been added by the authors as well: one for a subscriber type of sensor that listens to a ROS topic and one for utilizing a ROS service type. Those handlers are specific to the application and because we do not have an application where we need them, we have left them out. Also the initialization handler is left as a stub. It could be used to run start the launch file, but we decided it is better to leave the starting of the Gazebo simulator and running the hierarchical game decoupled, so the simulator can continue to run even if the hierarchical game is stopped.

5.2. Additional Problems

When testing our implementation of the handlers in combination with Gazebo, some additional problems had arisen.

For one, different from the basic simulation done by LTLMoP, the robot can get into situations where it cannot move, which can have several reasons. As an example, a dynamic obstacle, like a person could get in the way of the planned path and therefore make the robot replan its path or even stop.

Another reason came up because the world in Gazebo might differ slightly from the map in LTLMoP, since both are created manually. Although the same floor plans were used, pixel accurate drawing in LTLMoP with a computer mouse is hard to achieve, because it does not allow to zoom in or scale the map in another way. As a practical size we had chosen a resolution of $10\frac{\text{cm}}{\text{px}}$, thus, being off one or two pixels is an error of the size of the robot or a wall for example. This leads to situations where the coordinates translated from LTLMoP into the Gazebo world may be inside of a wall, so the robot will not be able to reach the goal.

Also the calculated translation matrix between the two coordinate systems adds some error if it is done by the calibration tool, because it is hard to get the exact position of a point in LTLMoP to the exact corresponding coordinates of the Gazebo simulation, even if the the accurate position of Gazebo is used, and not the approximated one by AMCL, because the robot has to be positioned manually and there might not be a good indicator where this position should be. In our examples we have used corners and doors of the building as points, so we had to move the robot inside of walls, which could also not be done in a real setting.

Not only dynamic obstacles like people could be an issue, but also static obstacles that are not modeled in the LTLMoP map. Examples could be desks or cupboards, which in principle could be moved, but mostly are not. Again, the goal coordinates could be inside of some obstacle.

To deal with the failures to move because of inaccuracies or obstacles, we have added a resolution function that tries to resolve the failure, as seen in Algorithm 3.

Algorithm 3 Handling blockages and failures

```
1: procedure RESOLVE
2:     Try different points in the region
3:     if all failed then
4:         Try to avoid the region (changing the specification of this layer)
5:         Rerun the game with new specification
```

The first point we try to reach in the next region is its center. Should that fail, because there is some obstacle or for some other reason, we first try to go to another point in the region. The next points tried are the shuffled corners of the region. Depending on the algorithm used as global planner this can lead to good results, meaning the region can be reached at some point and the hierarchical algorithm can continue.

Once we run out of points to reach for a region, we consider the goal as failed and communicate this failure up to the `LocalGame` instance of this game. It changes the specification, by adding the line *always not region*, where `region` is the failed region as reported by the motion handler. Then the specification is rewritten as a new file, which sets the `had_changes` flag in the `LocalGame` and leads to the new game getting synthesized and run if it did not exist already as seen in Algorithm 2. In case the new specification could not be synthesized, for example if the region we could not reach was the only exit to the next game, the next higher layer is informed about the failure. As mentioned in Section 4.1.4, this leads to the robot potentially visiting regions twice before going to the next region, so to avoid that memory propositions should be inserted.

Now that the general concept is described, the complete algorithm is shown in Algorithm 4. One important detail is that even if the `move_base` node reports that the goal was reached in line 19 we treat it as a failure, because we have checked if we are in the goal region before in line 2 and return early, which means the goal was not in the correct region after transforming the coordinates back to the LTLMoP system, indicating inaccuracies. In that case a `STATS` message is passed up to the top layer and logged to a file there, to see where issues in the scenario exist. When it is reported that the robot is failing to reach a region repeatedly, it might be better to mark the region as obstacle in the scenario, so that the robot will not try to go there in the first place.

To actually check if we are inside the goal region, we ask for the current position from the pose handler, transform these coordinates into the LTLMoP system and check if the transformed coordinates are inside of the polygon describing the region.

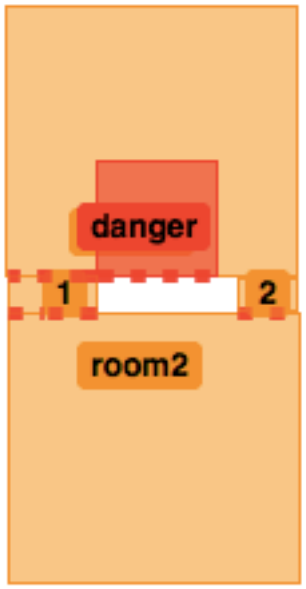
As we have mentioned before, results depend on the global planner used and generally the settings of the navigation stack. The default for the global planner is to use the Dijkstra algorithm to find a global plan for reaching the goal. The problem with this and our handling of failures to move in combination with the inaccuracies is that the global plan created might differ too much from the implicit plan given by the winning strategy created by LTLMoP and therefore can ignore the specification.

Consider a room that has two doors leading to the next room, like in Figure 5.3. The goal of the robot is the lower room, itself being in the upper room. The goal told by LTLMoP is the left door and the specification states the danger

Algorithm 4 MoveBaseMotionHandler

```
1: procedure gotoRegion(current_region, next_region)
2:     if we are in the next_region then
3:         return True
4:     prepare regions and names
5:     check if we are in another region
6:     if we have failed last iteration then
7:         if we have points left to try then
8:             get the next point to try
9:             send command to move_base
10:        else
11:            report failure to LocalGame
12:            exit
13:    else if our current goal is different from next_region then
14:        update our goal
15:        prepare list of points
16:        get the center of the region
17:        create goal and send to move_base
18:    get the status of move_base
19:    if status is succeeded then
20:        set failed to True
21:        return False
22:    else if status is aborted, rejected or lost then
23:        if if we have no points to try left then
24:            report failure to LocalGame
25:        set failed to True
26:        return False
27:    else if status is neither active nor pending then
28:        report that something else is going on
29:    return False
```

zone should be avoided. If the robot fails to move to the region 1 for some reason, the global planner might decide that it could reach the goal coordinates by going through the danger zone and using region 2 and therefore ignoring the specification.




```
robot starts in room1
always not danger
go to room2
```

Figure 5.3.: Example of rooms with two doors and a danger zone that should be avoided

We have considered two solutions to this problem. One is to stop the robot once it is in a region it is not supposed to be in, meaning neither the region it was starting from nor the region it was trying to reach. This could handled in the same way a normal failure to reach the region, by choosing a different point in the region and trying again.

Another solution specific to the global planner being too advanced is choosing another global planner. This is supported by the ROS nodes through configuration. Instead of using the global planner that implements the Dijkstra algorithm we chose the *CarrotPlanner*, which is a very basic planner, that only plans a straight line from the robot's current position to the goal. Since the global planning is done by LTLMoP already, this also seems to be the more suitable approach. Because it is still using a local planner, smaller obstacles can still be avoided, but it tends to report a failure more often than the more complex counterpart.

The configuration of these planners also allows to tweak the behaviour. In combination with the carrot planner, important settings for the local planner are goal tolerances, to set the with which distance to the goal it should be considered reached, the simulation time, where a higher value means the simulated trajectories are longer, and the trajectory scoring, where factors for the cost functions can be set. With the trajectory scoring settings, the bias for trajectories following the global plan or those which get closer to the local goal should be preferred. We have chosen the settings described in [25] as starting values and used it as a reference for the effects of different settings.

5.3. Experiments and observations

Although our initial goal was to run the algorithm on a real Turtlebot, we decided it was not feasible at that time. First, a detailed floor plan of the building would have to be acquired and copied over to an LTLMoP map, which could have lead to very imprecise regions. Secondly and more importantly, the building in question has a lot of glass walls, which can not be sensed with the sensors available on the Turtlebot, so the robot would run into them and also the position would be inaccurate, if the map available to the navigation module expects a (glass) wall, but does not sense it.

Therefore, to see how the implementation and execution does in the 3D simulated setting, we ran the building example with a minor modification in one room in combination with ROS. In order to see if the algorithm works if one door is blocked or closed, we have added another door from room 1 in building 1 to room 2.

At the start of the simulator we first moved the robot to some initial position and used the function to estimate its position with Rviz, so it has a good idea where it actually is. Then we have started the hierarchical games of the highest layer. This will get the current position of the robot according to the pose handler and determine the region it is in together with the next goal. It then starts the game of the next lower layer via the `AbstractHandler`.

As we have already mentioned before, different global planners can be used with the `move_base` package. In our example, the carrot planner had more trouble with navigating through doors and around obstacles, but this is mainly because the regions we have chosen were rather large and not well positioned for this use case. This lead to the robot trying to cut corners and fails to reach the goal with the carrot planner, as seen in Figure 5.4.

By defining regions, that are just as big as the doors, around doors, LTLMoP would send the robot to the center of the region above or below the door and therefore avoid the walls. Having more regions would also allow to treat ob-

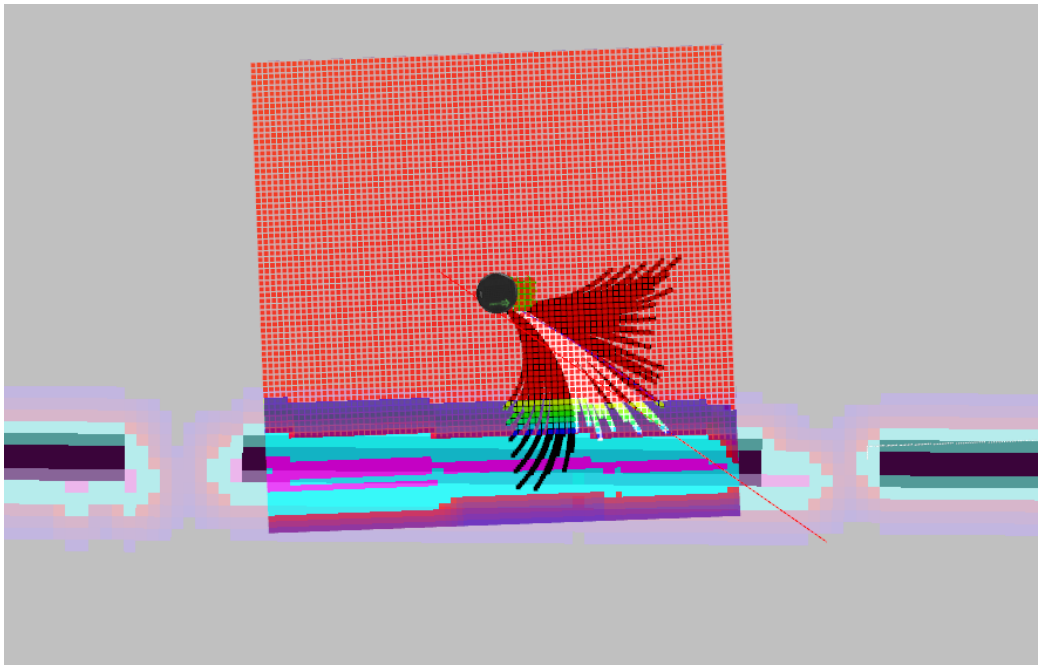


Figure 5.4.: Carrot Planner fails on door

stacles in a better way. Consider a region we should visit as per specification, but half of the region is occupied by some obstacle and only the other half is relevant. Then it might happen that the robot tries to move through the obstacle, fails, and thus cannot reach the goal. However, if those regions were split in two, the algorithm would change the specification to avoid the occupied region and find another way to the goal.

Another persisting problem when running the scenario is that due to the discussed inaccuracies, the pose handler might return coordinates, which are not inside of any region after they get translated to the LTLMoP coordinates. This is especially the case if the robot is close to walls. Also, because every game can have a different configuration and therefore different coordinate translation matrices it can also happen, that the layer that represents a building considers the robot being in room 1 and therefore starts a local game of room 1, but room 1 considers that position not in any of its regions.

While having different translation matrices might seem problematic, the overall accuracy of the lowest layer seemed to be better in our experience when compared to having the same translation matrix for every game. When we have tried using only one matrix, moving the robot through doors was troublesome, because the coordinates were often in or behind the walls to the next room. This is why we decided to use one calibration matrix for the higher levels and calibrate every room by itself.

In general, finding the exact pose of the robot in big, empty rooms is hard, because of the lack of features the algorithm could orientate to. If the static map only consists of the walls and the robot moves along those without having any other features, the pose will get more and more inaccurate because it takes into account that moving a certain amount of time with a specific velocity might not actually result in the same distance the robot actually travelled.

6. Conclusion and Outlook

During the course of this work, we have implemented the hierarchical algorithm proposed by Schmuck and Majumdar with slight modifications. It was done by integrating it into a tool called LTLMoP. We have discussed how we implemented the individual parts in more detail and closed the chapter by evaluating our implemented solution. The evaluation was done by studying the pros and cons of the hierarchical approach for controller synthesis with the traditional approach of monolithic games in context of the LTLMoP tool.

We have seen, that our implementation can have a big impact on the time needed to synthesize strategies, but also that it is very dependent on the scenario. The time and memory needed to find a strategy depends on the number of regions as well as the count of propositions used, thus reducing both in hierarchical games can lead to a better result in terms of resources used. In cases where for example the number of propositions cannot be reduced for every game, the overall time needed to synthesize all smaller strategies might be higher than in the monolithic version, but finding a strategy every individual game is relatively fast.

Then the problem has been lifted from a simple simulation to a more complex and realistic simulation using the 3D simulator Gazebo and the collection of libraries and tools around ROS, which provides state-of-the-art algorithms for several, more known problems in robotics. These have been integrated in our implementation of the hierarchical approach and experiments, if this is viable to use in a more realistic setting has been tested. We have tried to address additional problems that occurred when running in the more complex context, such as inaccuracies, obstacles and failures by retrying and changing the specification to avoid problematic regions.

Because ROS is targeted at real robots and not only 3D simulations, the step from the simulation to running it on actual robots is minor. The only changes needed would be to start ROS without the simulator, load the correct map for the navigation and run the LTLMoP instances on the robot itself or connect to the robot's ROS master node through the network.

Further development to improve the result could be done. As we have mentioned in Section 4.2, setting up the individual games for the hierarchical approach can be time consuming. The graphical interface of LTLMoP is not meant to edit several games at once, so as one improvement the interface could be up-

graded to allow the user to edit all local games inside of one instance of LTLMoP, by adding views for different levels of abstraction or defining subregions or refinements of games directly. This and the ability to scale or zoom the view in the region editor could improve the accuracy we have described in Section 5.2. By allowing the user to refine regions in a game of a lower level it could be asserted they have the exact same borders and with zooming the regions drawn by hand could be more precise, maybe even with the help of rulers known by graphic editing software or a way to create maps or regions inside of some external graphic editing software.

Another improvement for the lowest level of games would be the ability to define a grid of regions automatically. While this would be fairly easy for regions that are rectangular, this would be very helpful, because doing it manually is time consuming and repetitive.

Following the idea of a single instance of LTLMoP it could be nice to have a combined specification editor. Currently, every specification is local to a single game and similar to the idea of a combined region editor you could also implement a combined specification editor.

As far as the 3D simulation experiment is concerned, efforts to make it more realistic by adding sensor noise to the experiments could be made.

Algorithmically it could be interesting to explore the integration of a mapping algorithm like *Simultaneous Localization and Mapping* (SLAM) with the hierarchical approach. Static obstacles repeatedly detected by the sensors could be added as regions to be avoided through changing the specification permanently or maybe even the other way around, if obstacles disappear and the regions could be traversed in general.

Appendices

A. Source Code

Our modified and additional source code can be found on Github: `https://github.com/ayonix/LTLMoP/tree/hier_ros`, the branch is called `hier_ros`. The implementation of `LocalGame` can be found in *src/hierarchical.py*, `AbstractHandler` is in *src/lib/handlers/share/MotionControl/AbstractHandler.py* and the handlers for ROS are located in *src/lib/handlers/Hierarchical*.

A.1. Conventions

- File names: `basename.hierarchy_level.path_of_game.extension`, joined by periods, example: `example.0.3.2.spec`, in order to find the specification files. The basename should not any period.

- Goals: Use `go to` for goals, `visit` for temporary goals, both get translated to $\square \lozenge goal$, used to differentiate between the goals by another level and goals of the current level

- Exits: named `exit_level_from_to#identifier`, `#identifier` is optional if there is only one exit between regions `from` and `to`

- Regions: regions should not contain any characters but alphanumerical and underscores, since other characters can lead to problems if decomposing is turned off

A.2. Messages

Type	Data	Comment
Executor(l) → LocalGame(l)		
BORDER	region	Border to region is crossed
LocalGame(l) → AbstractHandler(l+1)		
BORDER	region	Border to region was crossed, passing it on
STATS	dict	Passing it on
UNSYNTH	region	We could not synthesize, tell the parent
AbstractHandler(l) → LocalGame(l)		
BORDER	region	Used for basic simulation only
STATS	dict	Passing it on
UNSYNTH	region	Lower game was unrealizable, just pass it on to this level's LocalGame
MoveBaseMotionHandler(l) → LocalGame(l)		
FAIL	region	We couldn't move to the region
STATS	dict	Reporting that move_base reached goal, but we were still in the wrong region

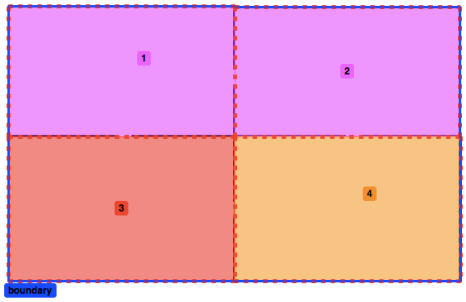
Table A.1.: Messages sent, listed by sender and receiver, the l indicates if it communicates with the instance of the same level, or the level above

B. Scenarios

The examples we have used can be found here: `https://github.com/ayonix/hierarchical_examples`.

B.1. Reorder

The idea in this scenario is that the higher level tells the lower level to move from region 1 to region 2, but the lower level cannot go there. It tells the upper level of its failure and the upper level therefore reorders the goals, to go to region 3 first. From region 3 it goes to region 4, where it cannot go to region 2 again and therefore stops, since region 2 was already the last goal in the upper level.

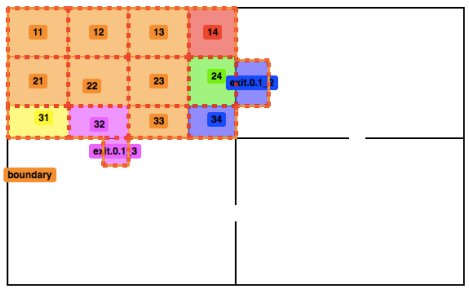


(a) reorder.1.regions

```
if not v1 then visit 1
if not v2 then visit 2
if not v3 then visit 3
if not v4 then visit 4

v1 is set on 1 and reset on false
v2 is set on 2 and reset on false
v3 is set on 3 and reset on false
v4 is set on 4 and reset on false
```

(b) reorder.1.spec

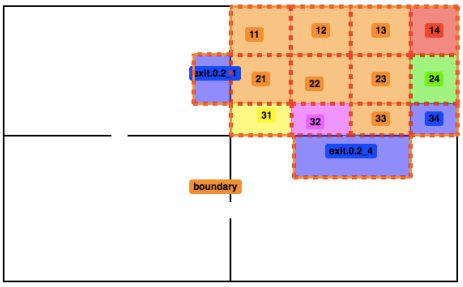


(a) reorder.0.1.regions

```
go to exit.0.1_2

always not exit.0.1_2
```

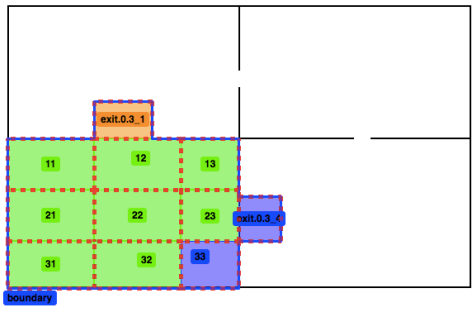
(b) reorder.0.1.spec



(a) reorder.0.2.regions

```
go to exit.0.2_1
```

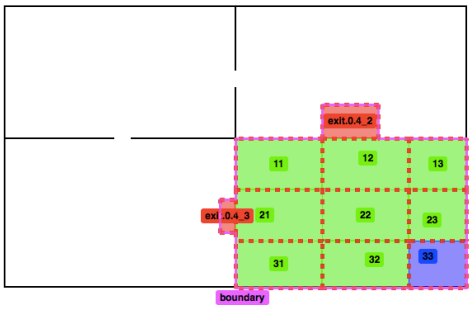
(b) reorder.0.2.spec



(a) reorder.0.3.regions

```
go to exit.0.3_4
```

(b) reorder.0.3.spec



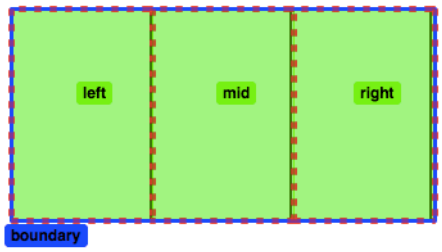
(a) reorder.0.4.regions

```
always not exit.0.4_2
go to exit.0.4_2
```

(b) reorder.0.4.spec

B.2. Lock because of reordering

In this scenario the live lock behaviour is shown if two regions are tried to be reached in vain.

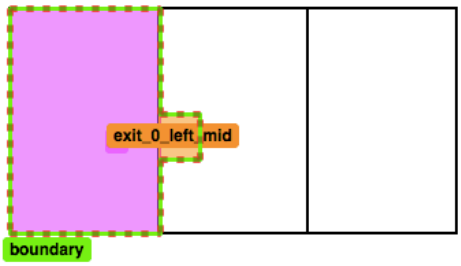


(a) lock.1.regions

```
if not vleft then visit left
if not vright then visit right

vleft is set on left and reset on false
vright is set on right and reset on false
```

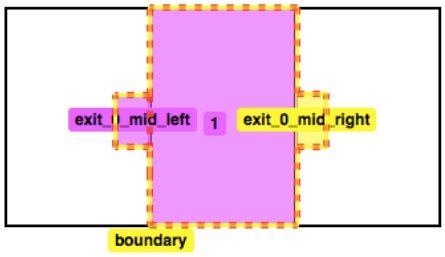
(b) lock.1.spec



(a) lock.left.regions

```
go to exit_0_left_mid
```

(b) lock.left.spec

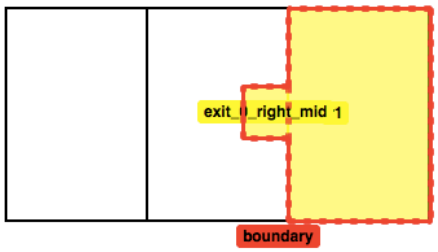


(a) lock.mid.regions

```
always not exit_0_mid_right
always not exit_0_mid_right

go to 1
```

(b) lock.mid.spec



(a) lock.right.regions

```
go to exit_0_right_mid
```

(b) lock.right.spec

B.3. Multiple doors example

Simulating the situation where a door is marked as blocked, but there is another door to the same region. The strategy for this scenario is to go to door2 directly.

(a) doors.regions

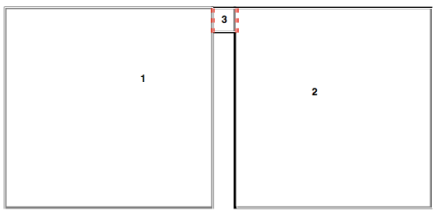
```
always not door1
go to door1 or door2
```

(b) doors.spec

B.4. Buildings with ROS

To test a scenario we have used the following specifications and regions.

B.4.1. Level 2

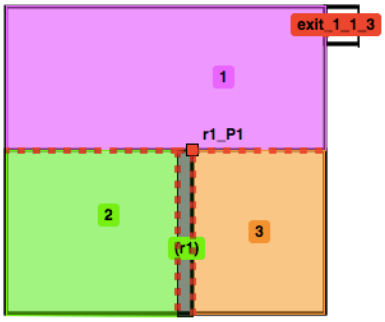


(a) buildings.2.regions

```
Group buildings is 1,2
visit all buildings
```

(b) buildings.2.spec

B.4.2. Level 1



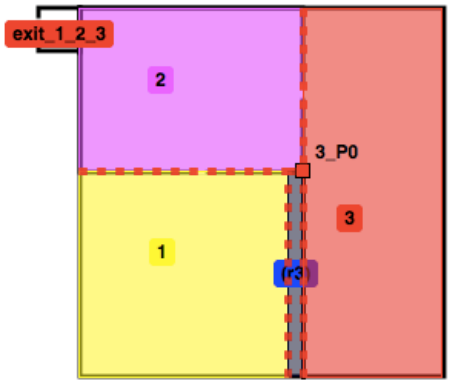
(a) buildings.1.1.regions

```
if not d1 then visit 1
if not d2 then visit 2
if not d3 then visit 3

d1 is set on 1 and reset on exit_1_1_3
d2 is set on 2 and reset on exit_1_1_3
d3 is set on 3 and reset on exit_1_1_3

go to exit_1_1_3
```

(b) buildings.1.1.spec

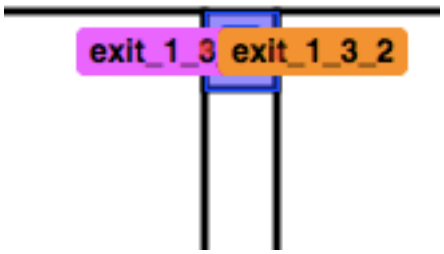


(a) buildings.1.2.regions

```
visit 1
visit 2
visit 3

go to exit_1_2_3
```

(b) buildings.1.2.spec

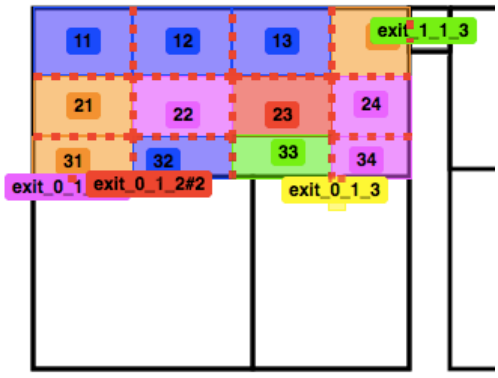


(a) buildings.1.3.regions

```
go to exit_1_3_2
```

(b) buildings.1.3.spec

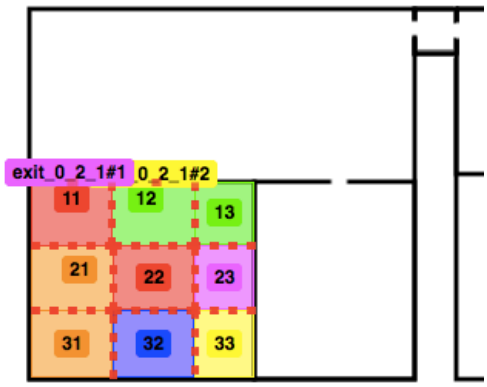
B.4.3. Level 0



(a) buildings.0.1.1.regions

```
go to exit_0_1_2#1
```

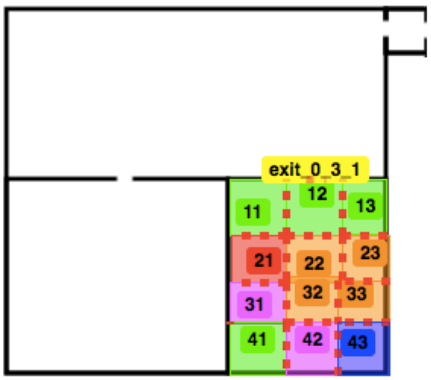
(b) buildings.0.1.1.spec



(a) buildings.0.1.2.regions

```
go to exit_0_1_2#1
```

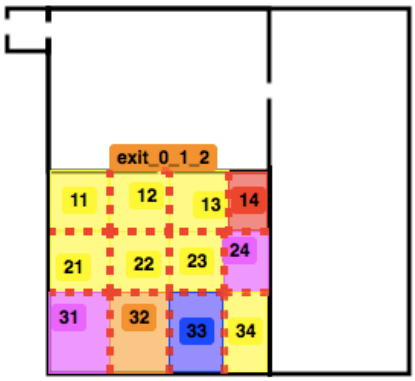
(b) buildings.0.1.2.spec



(a) buildings.0.1.3.regions

```
visit 43
go to exit_0_3_1
```

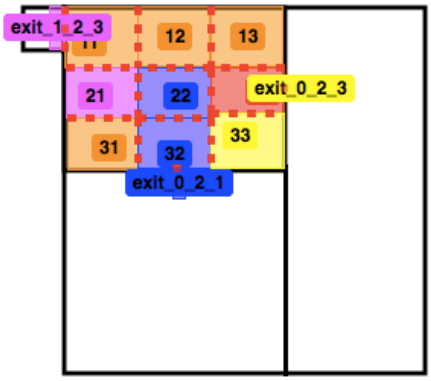
(b) buildings.0.1.3.spec



(a) buildings.0.2.1.regions

```
visit 31
go to exit_0_1_2
```

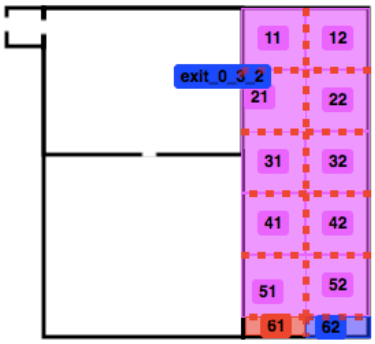
(b) buildings.0.2.1.spec



(a) buildings.0.2.2.regions

```
visit 22
go to 33
```

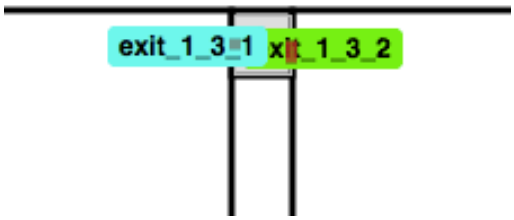
(b) buildings.0.2.2.spec



(a) buildings.0.2.3.regions

```
visit 31
go to 31
```

(b) buildings.0.2.3.spec



(a) buildings.0.3.1.regions

```
go to exit_1_3_2
```

(b) buildings.0.3.1.spec

Bibliography

[1] Benjamin Aminof, Fabio Mogavero, and Aniello Murano. "Synthesis of Hierarchical Systems". In: *Sci. Comput. Program.* 83 (Apr. 2014), pp. 56–79. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.07.001. URL: http://dx.doi.org/10.1016/j.scico.2013.07.001.

[2] M. Y. Belkhouche and B. Belkhouche. "Formal specification and simulation of the robot path planner". In: *2009 IEEE International Conference on Systems, Man and Cybernetics*. Oct. 2009, pp. 4484–4489. DOI: 10.1109/ICSMC.2009.5346910.

[3] Roderick Bloem et al. "Synthesis of Reactive(1) designs". In: *Journal of Computer and System Sciences* 78.3 (2012). In Commemoration of Amir Pnueli, pp. 911–938. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/j.jcss.2011.08.007. URL: http://www.sciencedirect.com/science/article/pii/S0022000011000869.

[4] Romain Brenguier, Jean-François Raskin, and Ocan Sankur. "Assume-Admissible Synthesis". In: *CoRR* abs/1507.00623 (2015). URL: http://arxiv.org/abs/1507.00623.

[5] Romain Brenguier et al. "AbsSynthe: abstract synthesis from succinct safety specifications". In: *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.* 2014, pp. 100–116. DOI: 10.4204/EPTCS.157.11. URL: http://dx.doi.org/10.4204/EPTCS.157.11.

[6] Rüdiger Ehlers and Vasumathi Raman. "Slugs: Extensible GR(1) Synthesis". In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 333–339. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_18. URL: http://dx.doi.org/10.1007/978-3-319-41540-6_18.

[7] Georgios E. Fainekos et al. "Temporal Logic Motion Planning for Dynamic Robots". In: *Automatica* 45.2 (Feb. 2009), pp. 343–352. ISSN: 0005-1098. DOI: 10.1016/j.automatica.2008.08.008. URL: http://dx.doi.org/10.1016/j.automatica.2008.08.008.

[8] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. "LTLMoP: Experimenting with language, Temporal Logic and robot control". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2010, pp. 1988–1993. DOI: 10.1109/IROS.2010.5650371.

[9] Open Source Robotics Foundation. *Gazebo*. URL: http://gazebosim.org (visited on 11/25/2016).

[10] Open Source Robotics Foundation. *ROS.org | Powering the world's robots*. URL: http://www.ros.org (visited on 11/25/2016).

[11] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Transactions on Robotics and Automation* 4 (1997), p. 1.

[12] Dieter Fox et al. "Monte carlo localization: Efficient position estimation for mobile robots". In: *AAAI/IAAI* 1999 (1999), pp. 343–349.

[13] *Grammar for the English to LTL Parser*. May 2011. URL: https://github.com/VerifiableRobotics/LTLMoP/raw/development/doc/grammar.pdf (visited on 11/16/2016).

[14] Gangyuan Jing, Rudiger Ehlers, and Hadas Kress-Gazit. "Shortcut through an evil door: Optimality of correct-by-construction controllers in adversarial environments". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Nov. 2013). DOI: 10.1109/iros.2013.6697048. URL: http://dx.doi.org/10.1109/IROS.2013.6697048.

[15] M. Kloetzer, X. C. Ding, and C. Belta. "Multi-robot deployment from LTL specifications with reduced communication". In: *2011 50th IEEE Conference on Decision and Control and European Control Conference*. Dec. 2011, pp. 4867–4872. DOI: 10.1109/CDC.2011.6160478.

[16] Manuel Mazo, Anna Davitian, and Paulo Tabuada. "PESSOA: A Tool for Embedded Controller Synthesis". In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 566–569. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_49. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_49.

[17] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) Designs". In: *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–380. ISBN: 978-3-540-31622-0. DOI: 10.1007/11609773_24. URL: http://dx.doi.org/10.1007/11609773_24.

[18] Amir Pnueli, Yaniv Sa'ar, and Lenore D. Zuck. "Jtlv: A Framework for Developing Verification Algorithms". In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 171–174. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_18. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_18.

[19] Rattanachai Ramaithitima et al. "Hierarchical Strategy Synthesis for Pursuit-Evasion Problems". In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*. 2016, pp. 1370–1378. DOI: 10.3233/978-1-61499-672-9-1370. URL: http://dx.doi.org/10.3233/978-1-61499-672-9-1370.

[20] Vasumathi Raman and Hadas Kress-Gazit. "Analyzing Unsynthesizable Specifications for High-Level Robot Behavior Using LTLMoP". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 663–668. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_54. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_54.

[21] Roni Rosner. "Modular Synthesis of Reactive Systems". PhD thesis. The Weizmann Institute of Science, Feb. 1991.

[22] I. Saha et al. "Automated composition of motion primitives for multi-robot systems from safe LTL specifications". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2014, pp. 1525–1532. DOI: 10.1109/IROS.2014.6942758.

[23] Anne-Kathrin Schmuck and Rupak Majumdar. "Dynamic Hierarchical Reactive Controller Synthesis". In: *CoRR* abs/1510.07246 (2015). URL: http://arxiv.org/abs/1510.07246.

[24] Kai Weng Wong, Cameron Finucane, and Hadas Kress-Gazit. "Provably-correct robot control with LTLMoP, OMPL and ROS". In: *IROS*. 2013, p. 2073.

[25] Kaiyu Zheng. *ROS Navigation Tuning Guide*. Sept. 2016. URL: http://www.zkytony.com/documents/navguide.pdf.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Kaiserslautern, den _____ _____

Ort, Datum *Adrian Leva*