LTLMoP Documentation

Release 0.9

Cameron Finucane, Gangyuan (Jim) Jing, Hadas Kress-Gazit

March 20, 2014

Contents:

CHAPTER 1

Applications

1.1 specEditor.py - Specification Editor

A development environment for specifications written in structured English, allowing for editing, compilation, and execution/simulation

class specEditor.**AnalysisResultsDialog**(*parent*, **args*, ***kwds*)

appendLog(*text*, *color='BLACK'*)

markFragments(*agent*, *section*, *jx=None*)

onButtonClose(*event*)

onClickRefineAnalysis(*event*)

populateTree(*gentree*)

class specEditor.**MapDialog**(*parent*, **args*, ***kwds*)
A simple little dialog that displays the regions on top of the map so that you can select a region visually instead of just choosing the name.

drawMap(*event*)

onMapClick(*event*)

class specEditor.**RedirectText**(*parent*, *aWxTextCtrl*)
A class that lets the output of a stream be directed into a text box.

http://mail.python.org/pipermail/python-list/2007-June/445795.html

write(*string*)

class specEditor.**SpecEditorFrame**(**args*, ***kwds*)
The main application window!

appendLog(*text*, *color='BLACK'*)

askIfUserWantsToSave(*action*)
Give the user the opportunity to save the current document.

'action' is a string describing the action about to be taken. If the user wants to save the document, it is saved immediately. If the user cancels, we return False.

From pySketch example

doClose(*event*)
Respond to the "Close" menu command.

drawLocMap(*event*)
Respond to a request to redraw the contents of the decomposed map

highlight(*l*, *type*)

highlightCores(*guilty*, *compiler*)

initializeNewSpec()

onClickEditRegions(*event*)

onImportRegion(*event*)
Ask the user for a region file and then import it.

onLocPhraseSelect(*event*)

onMapSelect(*event*)
Show the map with overlayed regions so that the user can select a region name visually.

onMenuAbout(*event*)

onMenuAnalyze(*event*)

onMenuCompile(*event*)

onMenuConfigSim(*event*)

onMenuCopy(*event*)

onMenuCut(*event*)

onMenuMopsy(*event*)

onMenuNew(*event*)
Create a new specification.

onMenuOpen(*event*)
Ask the user for a specification file to open, and then open it.

onMenuPaste(*event*)

onMenuQuit(*event*)

onMenuRedo(*event*)

onMenuSave(*event=None*)
If the file has been saved already, save it quietly. Else, ask for a filename and then save it.

onMenuSaveAs(*event*)
Ask the user for a filename to save the specification as, and then save it.

onMenuSetCompileOptions(*event*)

onMenuSimulate(*event*)
Run the simulation with current experiment configuration.

onMenuUndo(*event*)

onMenuViewAut(*event*)

onMouseDwellEnd(*event*)

onMouseDwellStart(*event*)

onPropAdd(*event*)
Display a dialog asking for a proposition name and then add it to the appropriate proposition list.

onPropRemove(*event*)
Remove the selected proposition from the list.

onPropToggle(*event*)

onPropositionDblClick(*event*)
Add the proposition name to the spec when you double click the name

onRegionLabelStyleChange(*event*)

onSpecTextChange(*event*)

onStyleNeeded(*e*)

openFile(*filename*)

refineAnalysis()

updateFromRFI()
Update the GUI to reflect a newly loaded region file

class specEditor.**WxAsynchronousProcessThread**(*cmd*, *callback*, *logFunction*)
Make sure callbacks from AsynchronousProcessThreads are safe for threading in wx by wrapping with a wx.CallAfter.

1.2 regionEditor.py - Region Editor

A simple polygonal region editor. The code is kind of a mess, but it does what it needs to do.

Based on pySketch by Erik Westra (ewestra@wave.co.nz) and Bill Baxter (wbaxter@gmail.com)

class regionEditor.**DrawingFrame**(*parent*, *id*, *title*, *fileName=None*)
A frame showing the contents of a single document.

askIfUserWantsToSave(*action*)
Give the user the opportunity to save the current document.

'action' is a string describing the action about to be taken. If the user wants to save the document, it is saved immediately. If the user cancels, we return False.

checkSubfaces(*obj*)

createPoly(*points*)
Create poly object.

createRect(*x*, *y*, *width*, *height*)
Create a new rectangle object at the given position and size.

deselectAll()
Deselect every Region in our document.

doChooseAddPtTool(*event=None*)
Respond to the "Poly Tool" menu command.

doChooseCalibPtTool(*event=None*)
Respond to the "Toggle Calibration Point Tool" menu command.

doChooseDelPtTool(*event=None*)
Respond to the "Poly Tool" menu command.

doChoosePolyTool(*event=None*)
Respond to the "Poly Tool" menu command.

doChooseRectTool(*event=None*)
Respond to the "Rect Tool" menu command.

doChooseSelectTool(*event=None*)
Respond to the "Select Tool" menu command.

doClose(*event*)
Respond to the "Close" menu command.

doDelete(*event=None*)
Respond to the "Delete" menu command.

doDuplicate(*event*)
Respond to the "Duplicate" menu command.

doEditRegion(*event=None*)
Respond to the "Edit Region" menu command.

doExit(*event*)
Respond to the "Quit" menu command.

doMakeBoundary(*event*)

doNew(*event*)
Respond to the "New" menu command.

doOpen(*event*)
Respond to the "Open" menu command.

doRevert(*event*)
Respond to the "Revert" menu command.

doSave(*event=None*)
Respond to the "Save" menu command.

doSaveAs(*event*)
Respond to the "Save As" menu command.

doSelectAll(*event*)
Respond to the "Select All" menu command.

doSetBackground(*event=None*)

doShowAbout(*event*)
Respond to the "About Region Editor" menu command.

doUndo(*event*)
Respond to the "Undo" menu command.

drawRegions(*dc*, *pdc*, *drawLabels=True*, *drawAdjacencies=True*)

getProjectDir()
Make sure we've saved our region file somewhere so that our project directory and project name are defined.

loadContents()
Load the contents of our document into memory. Note that the drawing order of regions is last to first.

onDoubleClickEvent(*event*)
Respond to a double-click within our drawing panel.

onKeyEvent(*event*)
Respond to a keypress event.

We make the arrow keys move the selected object(s) by one pixel in the given direction.

onMouseEvent(*event*)
Respond to the user clicking on our main drawing panel.

How we respond depends on the currently selected tool.

onPaintEvent(*event*)
Respond to a request to redraw the contents of our drawing panel.

onRightClick(*event*)
Respond to the user right-clicking within our drawing panel.

We select the clicked-on item, if necessary, and display a pop-up menu of available options which can be applied to the selected item(s).

onTimerEvent(*event*)

onToolIconClick(*event*)
Respond to the user clicking on one of our tool icons.

recalcAdjacency()
Call the RegionFileInterface's recalcAdjacency() method to figure out where to draw dotted transition lines

saveContents()
Save the contents of our document to disk.

select(*obj*)
Select the given Region within our document.

selectAll()
Select every Region in our document.

selectByRectangle(*x*, *y*, *width*, *height*)
Select every Region in the given rectangular region.

selectMany(*objs*)
Select the given list of Regions.

testBetween()

testNear()

class regionEditor.**EditRegionDialog**(*parent*, *title*)
Dialog box used to edit the properties of a region.

The user can edit the region's name and color.

dialogToObject(*obj*)
Copy the properties from the dialog box into the given text object.

objectToDialog(*obj*)
Copy the properties of the given text object into the dialog box.

class regionEditor.**ExceptionHandler**
A simple error-handling class to write exceptions to a text file.

Under MS Windows, the standard DOS console window doesn't scroll and closes as soon as the application exits, making it hard to find and view Python exceptions. This utility class allows you to handle Python exceptions in a more friendly manner.

write(*s*)
Write the given error message to a text file.

Note that if the error message doesn't end in a carriage return, we have to buffer up the inputs until a carriage return is received.

class regionEditor.**SketchApp**
The main pySketch application object.

OnInit()
Initialise the application.

class regionEditor.**TextObjectValidator**
This validator is used to ensure that the user has entered something into the text object editor dialog's text field.

Clone()
Standard cloner.

Note that every validator must implement the Clone() method.

TransferFromWindow()
Transfer data from window to validator.

The default implementation returns False, indicating that an error occurred. We simply return True, as we don't do any data transfer.

TransferToWindow()
Transfer data from validator to window.

The default implementation returns False, indicating that an error occurred. We simply return True, as we don't do any data transfer.

Validate(*win*)
Validate the contents of the given text control.

class regionEditor.**ToolPaletteIcon**(*parent*, *iconID*, *iconName*, *toolTip*)
An icon appearing in the tool palette area of our sketching window.

Note that this is actually implemented as a wx.Bitmap rather than as a wx.Icon. wx.Icon has a very specific meaning, and isn't appropriate for this more general use.

deselect()
Deselect the icon.

The icon's visual representation is updated appropriately.

select()
Select the icon.

The icon's visual representation is updated appropriately.

regionEditor.**main**()
Start up the pySketch application.

class configEditor.**addRobotDialog**(*parent*, **args*, ***kwds*)

onChangeHandler(*event*)

onChooseRobot(*event*)

onClickConfigure(*event*)

onClickOK(*event*)

onEditRobotName(*event*)

configEditor.**drawParamConfigPane**(*target*, *method*, *proj*)

class configEditor.**handlerConfigDialog**(*parent*, **args*, ***kwds*)

onClickDefaults(*event*)

class configEditor.**propMappingDialog**(*parent*, **args*, ***kwds*)

onClickApply(*event*)

onClickMapping(*event*)

onClickOK(*event*)

onEditMapping(*event*)

onSelectHandler(*event*)

onSelectProp(*event*)

onSelectRobot(*event*)

class configEditor.**regionTagsDialog**(*parent*, **args*, ***kwds*)

onCheckRegion(*event*)

onClickAddTag(*event*)

onClickRemoveTag(*event*)

onClickTag(*event*)

class configEditor.**simSetupDialog**(**args*, ***kwds*)

doClose(*event*)

onCheckProp(*event*)

onClickAddRobot(*event*)

onClickApply(*event*)

onClickCancel(*event*)

onClickConfigureRobot(*event*)

onClickEditMapping(*event*)

onClickEditRegionTags(*event*)

onClickOK(*event*)

onClickRemoveRobot(*event*)

onConfigDelete(*event*)

onConfigImport(*event*)

onConfigNew(*event*)

onSetMainRobot(*event*)

onSimLoad(*event*)

onSimNameEdit(*event*)

1.3 execute.py - Top-level hybrid controller executor

This module executes a hybrid controller for a robot in a simulated or real environment.

> **Usage** `execute.py [-hn] [-p listen_port] [-a automaton_file] [-s spec_file]`

- The controlling automaton is imported from the specified `automaton_file`.
- The supporting handler modules (e.g. sensor, actuator, motion control, simulation environment initialization, etc) are loaded according to the settings in the config file specified as current in the `spec_file`.
- If no port to listen on is specified, an open one will be chosen randomly.
- Unless otherwise specified with the `-n` or `--no_gui` option, a status/control window will also be opened for informational purposes.

class `execute.`**`LTLMoPExecutor`**

This is the main execution object, which combines the synthesized discrete automaton with a set of handlers (as specified in a .config file) to create and run a hybrid controller

`initialize`(*spec_file*, *strategy_file*, *firstRun=True*)

Prepare for execution, by loading and initializing all the relevant files (specification, map, handlers, strategy) If *firstRun* is true, all handlers will be imported; otherwise, only the motion control handler will be reloaded.

`isRunning`()

return whether the automaton is currently executing

`loadAutFile`(*filename*)

This function loads the the .aut/.bdd file named filename and returns the strategy object. filename (string): name of the file with path included

`loadSpecFile`(*filename*)

`pause`()

pause execution of the automaton

`postEvent`(*eventType*, *eventData=None*)

Send a notice that an event occurred, if anyone wants it

`registerExternalEventTarget`(*address*)

`resume`()

start/resume execution of the automaton

`run`()

`shutdown`()

class `execute.`**`RedirectText`**(*event_handler*)

`flush`()

`write`(*message*)

`execute.`**`execute_main`**(*listen_port=None*, *spec_file=None*, *aut_file=None*, *show_gui=False*)

`execute.`**`usage`**(*script_name*)

Print command-line usage information.

1.4 calibrate.py - A tool for finding the transformation between map and real coordinates

This script helps you experimentally determine the coordinate transformation between points on your region map and points in your localization system.

The specific points used for calibration are chosen in the Region Editor.

Usage `calibrate.py [spec_file]`

class `calibrate.`**CalibrateApp**

OnInit()

class `calibrate.`**CalibrateFrame**(**args*, ***kwds*)

doCalibration()

moveRobot(*event, state=[False]*)

onButtonGo(*event*)

onPaint(*event=None*)

onResize(*event*)

setStepInfo(*label*, *button*)

1.5 simGUI.py - Experiment Monitor GUI

A basic user interface for watching the state of the robot during simulation/experiment, and pausing/resuming execution.

class `simGUI.`**SimGUI_Frame**(**args*, ***kwds*)

appendLog(*text*, *color='BLACK'*)

handleEvent(*eventType*, *eventData*)
Processes messages from the controller, and updates the GUI accordingly

initDialogue()

loadRegionFile(*filename*)

loadSpecFile(*filename*)

onClose(*event*)

onEraseBG(*event*)

onPaint(*event=None*)

onResize(*event=None*)

onSLURPSubmit(*event*)

onSimClear(*event*)

onSimExport(*event*)
Ask the user for a filename to save the Log as, and then save it.

onSimStartPause(*event*)

saveFile(*fileName*)
Write all data out to a file.

CHAPTER 2

Shared Modules

2.1 project.py - Abstraction layer for project files

This module exposes an object that allows for simplified loading of the various files included in a single project.

class `project.`**Project**
A project object.

determineEnabledPropositions()
Populate lists `all_sensors`, `enabled_sensors`, etc.

getCoordMaps()
Returns forward (map->lab) and reverse (lab->map) coordinate mapping functions, in that order

getFilenamePrefix()
Returns the full path of most project files, minus the extension.

For example, if the spec file of this project is `/home/ltlmop/examples/test/test.spec` then this function will return `/home/ltlmop/examples/test/test`

getStrategyFilename()
Returns the full path of the file that should contain the strategy for this specification.

importHandlers(*all_handler_types=None*)
Figure out which handlers we are going to use, based on the different configurations file settings Only one motion/pose/drive/locomotion handler per experiment Multiple init/sensor/actuator handlers per experiment, one for each robot (if any) Load in specified handlers. If no list is given, *all* handlers will be loaded. Note that the order of loading is important, due to inter-handler dependencies.

loadConfig(*name=None*)
Load the config object with name `name` (case-insensitive). If no name is specified, load the one defined as currently selected.

loadProject(*spec_file*)
Because the spec_file contains references to all other project files, this is all we need to know in order to load everything in.

loadRegionFile(*decomposed=False*)
Returns a Region File Interface object corresponding to the regions file referenced in the spec file

loadRegionMapping()
Takes the region mapping data and returns region mapping dictionary.

loadSpecFile(*spec_file*)

setSilent(*silent*)

writeSpecFile(*filename=None*)

project.**get_ltlmop_root**()

2.2 regions.py - Regions Module

A simple module that defines a class for describing and manipulating rectangular and polygonal regions.

This is completely free software; please feel free to adapt or use this in any way you like.

Some parts extracted from pySketch by Erik Westra (ewestra@wave.co.nz)

class regions.**Color**(*red=0*, *green=0*, *blue=0*)

Blue()

Green()

Red()

SetFromName(*name*)

class regions.**Point**(*x*, *y*)

class regions.**Region**(*type=1*, *position=Point(0.000000, 0.000000)*, *size=Point(0.000000, 0.000000)*, *height=0*, *color=None*, *points=None*, *name=''*)

A rectangular or polygonal region, defined by the following properties:

- 'name' Region name
- 'type' What type of region this is (rect or poly)
- **'position' The position of the object within the document** (i.e., the top-left corner of the region's bounding-box)
- 'size' The size of the object's bounding box
- 'color' Color to use for drawing the region
- 'pointArray' Polygon points, relative to region position (stored in CW order)
- 'alignmentPoints' True/False array indicating for each vertex whether or not to use it as an alignment point
- 'isObstacle' Boolean value indicating whether the region should be treated as an obstacle

NOTE: All coordinates are stored internally with (0,0) at the top left. X increases to right, and Y increases downwards.

addPoint(*point*, *index*)

Insert a new point at a given index, converting object type if necessary

faceAndFaceIntersection(*face1*, *face2*)

findPointsNear(*face*, *center*, *distance*)

findRegionNear(*distance*, *mode='underEstimate'*, *name='newRegion'*)

Given a region object and a distance value, return a new region object which covers the area that is within the 'distance' away from the given region.

getCenter()

Find the 'center' of a region

getData()
Return a copy of the object's internal data. This is used for undo and to save this region to disk.

getDirection()
Determine convexity/concavity and the order of the points stored in the pointArray. For details on the algorithm, please refer to:

http://local.wasp.uwa.edu.au/~pbourke/geometry/clockwise/index.html

getFaces(*includeHole=False*)
Wrapper function to allow for iteration over faces of regions. A face is a frozenset of the two points (in absolute coordinates) that make up the face.

getPoints(*relative=False*, *hole_id=None*)
Wrapper function to allow for iteration over the points of a region without worrying about whether it's a RECT or POLY. When hole_id is None, the boundary points of the region will be returned When Otherwise the points of holeList[hole_id] will be returned

getSelectionHandleContainingPoint(*x*, *y*, *boundFunc=None*)
Return the selection handle containing the given point, if any.

We return one of the predefined selection handle ID codes (defined at top).

objectContainsPoint(*x*, *y*)
Returns True iff this object contains the given point.

This is used to determine if the user clicked on the object.

objectWithinRect(*x*, *y*, *width*, *height*)
Return True iff this object falls completely within the given rect.

polyContainsPoint(*poly_pts*, *x*, *y*)

recalcBoundingBox()
Recalculate the bounding box for our object

removePoint(*index*)
Remove the point at the given index, converting object type if necessary

setData(*data*)
Set the object's internal data.

'data' is a copy of the object's saved data, as returned by getData() above. This is used for undo and to restore a previously saved region.

setDataOld(*data*)
Set the object's internal data.

'data' is a copy of the object's saved data, as returned by previous version of region editor. This function is only for backwards compatibility and will eventually be removed.

class regions.**RegionFileInterface**(*background='None'*, *regions=None*, *transitions=None*)
A wrapper class for handling collections of regions and associated metadata.

Overview of exposed data structures:

•background (string): relative path of background image file

•regions (list): list of Region objects, with properties defined below

•**transitions (list of lists):**

- key1 = Region object index
- key2 = Region object index

- values = Lists of faces connecting the two regions

getBoundingBox()

getCalibrationPoints()

getExternalFaces()
Returns a list of faces that are not connected to any other region on the map.

Generally, this only makes sense for maps that have already been decomposed.

getMaximumHeight()

getNextAvailableRegionNumber()
Look for the smallest region name of form r1, r2, ... available.

indexOfRegionWithName(*name*)

readFile(*filename*)
For file format information, refer to writeFile() above.

recalcAdjacency()
Calculate the region adjacency matrix and a list of shared faces

Returns a list of shared faces

setToDefaultName(*region*)

splitSubfaces(*obj1*, *obj2*)
If we have a face of region obj1 that overlaps with the face of another region obj2, (i.e. the obj1 face is collinear with and has at least one point on the obj2 face) we split the larger face into two or three parts as appropriate.

writeFile(*filename*)
File format is described inside the comments variable.

class regions.**Size**(*w*, *h*)

GetHeight()

GetWidth()

regions.**findRegionBetween**(*regionA*, *regionB*, *name='newRegion'*)
Find the region between two given regions (doesn't include the given regions)

regions.**pointLineIntersection**(*pt1*, *pt2*, *test_pt*)
Given two points (pt1, pt2), find the point on the line formed by those points that is nearest to test_pt and give the distance.

class regions.**prettierJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Subclass of JSONEncoder that stops indenting after 2 levels; only seems to work well with python2.6 version, not 2.7 :(

2.3 fileMethods.py - File Access Methods

Some routines for reading and writing plaintext config/data files, shared throughout the toolkit.

fileMethods.**properCase**(*str*)
Returns a copy of a string, with the first letter capitalized and all others lower-case (*CURRENTLY DOES NOTHING*, and I'm not sure why that change was made)

fileMethods.**readFromFile**(*fileName*)
A simple method for reading in data from text files of the following format:

```
========= SECTION1 ==========

# COMMENT

HEADER1: # COMMENT
DATA1
DATA2

HEADER2: # COMMENT
DATA3

========= SECTION2 ===========

HEADER1:
DATA1
```

Given a file with the above contents, the function will return the following dictionary of dictionaries:

```
{ 'SECTION1' : { 'HEADER1': ['DATA1', 'DATA2'],
                 'HEADER2': ['DATA3'] },
{ 'SECTION2' : { 'HEADER1': ['DATA1'] } }
```

NOTE:

- Headers *must* be followed by a colon
- Section names must have at least one equals sign on both sides
- Dictionary keys will always be normalized so that only the first letter is capitalized (*CURRENTLY DISABLED*)
- All items are returned as strings and should be cast to the appropriate type before use, if appropriate
- Lines beginning with # are treated as comments
- Blank lines at the end of sections are ignored
- Any data without a section title will be returned under the empty string key: `''`
- If no section titles are present, the outer dictionary will be ommitted

fileMethods.**writeToFile**(*fileName*, *data*, *comments={}*)
A simple method for writing data to text files of the format described in the `readFromFile()` function above.

All data will be output in key-sorted order for the sake of consistency.

Any items in the comments hash whose keys match those of the data hash will be included as comments in the appropriate section of the file.

If it exists, the comment keyed as `"FILE_HEADER"` will be added to the top of the output file.

2.4 parseEnglishToLTL.py - Structured English to LTL Translator

Module that parses a set of structured English sentences into the corresponding LTL subformulas.

parseEnglishToLTL.**bitEncoding**(*numRegions*, *numBits*)
This function creates a dictionary that contains the bit encoding for the current and next region. Takes number of regions and returns a dictionary with 'current' and 'next' as keys, each containing a list of the respective encodings.

parseEnglishToLTL.**createStayFormula**(*regionNames*, *use_bits=True*)

parseEnglishToLTL.**nextify**(*p*)

parseEnglishToLTL.**parseAfterEachTime**(*Cond*, *Requirement*, *sensorProp*, *allRobotProp*, *lineInd*, *StayFormula*)

parseEnglishToLTL.**parseCond**(*condition*, *sensorList*, *allRobotProp*, *ReqType*, *lineInd*)
This function creates the LTL formula representing the condition part of a conditional. It takes the condition and PropList - a list of propositions (to check that only 'legal' propositions are used)and 'lineInd' that indicates which line is being processed. Returns the LTL formula as a string.

parseEnglishToLTL.**parseConditional**(*Condition*, *ReqFormulaInfo*, *CondType*, *sensorList*, *allRobotProp*, *lineInd*)
This function creates the LTL formula representing a conditional. It takes the condition, the requirement formula (that was already parsed), the condition type, and the list of all propositions (to check that only 'legal' propositions are used) and 'lineInd' that indicates which line is being processed. Returns a dictionary with 2 keys: 'formula' containing the LTL formula as a string and 'type' containing the type of the requirement.

parseEnglishToLTL.**parseEvent**(*EventProp*, *SetEvent*, *ResetEvent*, *sensorProp*, *RobotProp*, *lineInd*)
This function creates the LTL formulas encoding when a proposition should be true and when false. This is used as a macro to define 'memory' propositions. It takes the proposition, the boolean formulas defining the set and reset events, the propositions (to check that only 'legal' propositions are used) and 'lineInd' that indicates which line is being processed. Returns the LTL formula as a string.

parseEnglishToLTL.**parseInit**(*sentence*, *PropList*, *lineInd*)
This function creates the LTL formula representing the initial conditions. It takes the sentence and PropList - a list of propositions (to check that only 'legal' propositions are used) and 'lineInd' that indicates which line is being processed. Returns the LTL formula as a string.

parseEnglishToLTL.**parseLiveness**(*sentence*, *sensorList*, *allRobotProp*, *lineInd*)
This function creates the LTL formula representing a basic liveness requirement. It takes the sentence, the sensor list and the list of all robot propositions (to check that only 'legal' propositions are used and to determine whether it is an environment safety or a robot one) and 'lineInd' that indicates which line is being processed. Returns a dictionary with 2 keys: 'formula' containing the LTL formula as a string and 'type' containing either 'EnvGoals' or 'SysGoals'.

parseEnglishToLTL.**parseSafety**(*sentence*, *sensorList*, *allRobotProp*, *lineInd*)
This function creates the LTL formula representing a basic safety requirement. It takes the sentence, the sensor list and the list of all robot propositions (to check that only 'legal' propositions are used and to determine whether it is an environment safety or a robot one) and 'lineInd' that indicates which line is being processed. Returns a dictionary with 2 keys: 'formula' containing the LTL formula as a string and 'type' containing either 'EnvTrans' or 'SysTrans'.

parseEnglishToLTL.**parseToggle**(*EventProp*, *ToggleEvent*, *sensorProp*, *RobotProp*, *lineInd*)
This function creates the LTL formulas encoding when a proposition's value should toggle (T->F, F->T). It takes the proposition, the boolean formula defining the toggle event, the propositions (to check that only 'legal' propositions are used) and 'lineInd' that indicates which line is being processed. Returns the LTL formula as a string.

parseEnglishToLTL.**replaceLogicOp**(*formula*)
This function replaces the logic operators with TLV convention.

parseEnglishToLTL.**replaceRegionName**(*formula*, *bitEncode*, *regionList*)
This function replaces the region names with the appropriate bit encoding.

parseEnglishToLTL.**writeSpec**(*text*, *sensorList*, *regionList*, *robotPropList*)
This function creates the Spec dictionary that contains the parsed LTL subformulas. It takes the text that contains the structured English, the list of sensor propositions, the list containing the region names and the list of robot propositions (other than regions).

2.5 createJTLVinput.py - LTL Pre-Processor Routines

Module that creates the input files for the JTLV based synthesis algorithm. Its functions create the skeleton .smv file and the .ltl file which includes the topological relations and the given spec.

createJTLVinput.**createInitialRegionFragment**(*regions*, *use_bits=True*)

createJTLVinput.**createLTLfile**(*fileName*, *spec_env*, *spec_sys*)
This function writes the LTL file. It encodes the specification and topological relation. It takes as input a filename, the list of the sensor propositions, the list of robot propositions (without the regions), the adjacency data (transition data structure) and a specification

createJTLVinput.**createNecessaryFillerSpec**(*spec_part*)
Both assumptions guarantees need to have at least one each of initial, safety, and liveness. If any are not present, create trivial TRUE ones.

createJTLVinput.**createSMVfile**(*fileName*, *sensorList*, *robotPropList*)
This function writes the skeleton SMV file. It takes as input a filename, the number of regions, the list of the sensor propositions and the list of robot propositions (without the regions).

createJTLVinput.**createTopologyFragment**(*adjData*, *regions*, *use_bits=True*)

createJTLVinput.**flattenLTLFormulas**(*f*)

class asyncProcesses.**AsynchronousProcessThread**(*cmd*, *callback*, *logFunction*)

kill()

run()

class decomposition.**decomposition**(*polygon*, *holes=[]*)

MP5()

calcAngle(*a*, *b*, *c*)
A function that calculates the angle between 0 and 2*pi rad swept by a counterclockwisr rotation from line segment ba to bc

a,b,c are vertices Return True when the angle is smaller than or equals pi Return False when the angle is larger than pi .

checkNextPoly(*allVertices*)

checkPointInside(*allVertices*)
True: There is at least one vertex inside the polygon generated False: There is no vertex inside

drawPoly(*polyList*, *fileName*)

findInitialVertex(*allVertices*)
Find notch vertex to start

allVertices: List of all vertices of the polygon to be decomposed

getFaces(*poly*)
Wrapper function to allow for iteration over faces of regions. A face is a tuple of the two points (in absolute coordinates) that make up the face, sorted so that a given face is defined uniquely.

FIXME: Make sure we take advantage of this uniqueness elsewhere; I think we check too many conditions sometimes

getVertices(*poly*)

lineLineIntersection(*l1p1*, *l1p2*, *l2p1*, *l2p2*)

linePolyIntersection(*poly*, *vertexA*, *vertexB*, *boundaryPoly*)

mergeHole(*allVertices*, *initialIndex*, *holeIndex*, *vertexIndex*)

pointPointDistance(*pt1*, *pt2*)

removeContour(*contour*)

reversePolyOrientation(*poly*)

class decomposition.**myVertex**(*x=0*, *y=0*)

decomposition.**removeDuplicatePoints**(*points*)

2.6 handlerSubsystem.py - Interface for working with handlers, configs

This module Defines objects for handler system. It also provides interface that deal with handler and config files

class handlerSubsystem.**ConfigFileParser**(*config_path*, *handler_path*, *proj*, *handler_dic=None*)
A parser loads all configuration files

loadAllConfigFiles()

loadConfigFile(*fileName*)

saveAllConfigFiles()

class handlerSubsystem.**ConfigObject**
A config file object!

getRobotByName(*name*)

saveConfig()
Save the config object. Return True for successfully saved, False for not

class handlerSubsystem.**HandlerObject**
A handler object!

fullPath(*robotName*, *configObj*)
Return the full path for this handler object for importing

getMethodByName(*name*)

getType()

setType(*h_type*)

toString(*forsave=True*)
Return the string representation of the handler object

forsave is True then the string is for saving the config file False is for initiate this handler during execution

class handlerSubsystem.**HandlerParser**(*path*)
A parser loads all handler information

loadAllHandlers()
Load all handlers in the handler folder

loadHandler(*folder*, *onlyLoadInit=False*)
Load all handler files within the given folder If onlyLoadInit is True, only the info of __init__ method will be loaded If over_write_h_type is given, then over write the handler type with it

return a list of handler objects

parseHandlers(*handlerFile*, *h_type*, *onlyLoadInit=False*, *over_write_h_type=None*)
Load method info (name,arg...) in the given handler file If onlyLoadInit is True, only the info of __init__ method will be loaded If over_write_h_type is given, then over write the handler type with it

returns a handler object or None if fail to load the given handler file

printHandler()

class handlerSubsystem.**HandlerSubsystem**(*proj*)
Interface dealing with configuration files and hanlders

constructMethodString(*robotName*, *handlerName*, *methodName*, *para_info*)
returns the string used to execute the corresponding method

getHandler(*htype*, *hname*, *rname=None*)

getRobotByType(*t*)
Assume only one robot is loaded per type

importHandlers(*configObj*, *all_handler_types*)
Figure out which handlers we are going to use, based on the different configurations file settings Only one motion/pose/drive/locomotion handler per experiment Multiple init/sensor/actuator handlers per experiment, one for each robot (if any) Note that the order of loading is important, due to inter-handler dependencies.

loadAllConfigFiles()

loadAllHandlers()

loadAllRobots()

method2String(*methodObj*, *robotName=''*)
Return the string representation according to the input method object

string2Method(*method_string*)
Return the method object according to the input string

class handlerSubsystem.**MethodObject**
A method object Each object represents one method of a given handler

getParaByName(*name*)

class handlerSubsystem.**ParameterObject**(*para_name=''*)
A parameter object Each object represents one parameter of a given method

getValue()

resetValue()

setValue(*value*)
This function makes sure all parameter are set according to the desired type

class handlerSubsystem.**RobotFileParser**(*path*, *handler_dic=None*)
A parser load robot file in each robotFolder

loadAllRobots()
Load all robot files in the handlers/robots folder

loadRobotData(*robot_data*)
Given a dictionary of robot handler information, returns a robot object holding all the information The dictionary is in the format returned by the readFromFile function If the necessary handler of the robot is not specified or can't be loaded, return None

loadRobotFile(*fileName*)
Given a robot file, return a dictionary holding its information

class handlerSubsystem.**RobotObject**(*r_name=''*, *r_type=''*, *driveH=None*, *initH=None*, *locoH=None*, *motionH=None*, *poseH=None*, *sensorH=None*, *actuatorH=None*)
A Robot object

class mapRenderer.**DrawableRegion**(**args*, ***kwds*)
Extends the Region class to allow drawing.

draw(*dc*, *pdc*, *selected*, *scale=1.0*, *showAlignmentPoints=True*, *highlight=False*, *deemphasize=False*)
Draw this Region into our window.

'dc' is the device context to use for drawing. If 'selected' is True, the object is currently selected and should be drawn as such.

classmethod fromRegion(*region*)

mapRenderer.**drawMap**(*target*, *rfi*, *scaleToFit=True*, *drawLabels=True*, *highlightList=[]*, *deemphasizeList=[]*, *memory=False*, *showBits=False*)
Draw the map contained in the given RegionFileInterface onto the target canvas.

class parseLP.**parseLP**
A parser to parse the locative prepositions in specification

checkOverLapping()
Check if and regions overlap each other Break the ones that overlap into portions that don't overlap

decomp()
Decompose the region with holes or are concave

decomposeWithOverlappingPoint(*polygon*)
When there are points overlapping each other in a given polygon First decompose this polygon into sub-polygons at the overlapping point

drawAllPortions()
Output a drawing of all the polygons that stored in self.portionOfRegion, for debug purpose

generateNewRegion()
Generate new regions for locative prepositions

intAllPoints(*poly*)
Function that turn all point coordinates into integer Return a new polygon

main(*argv*)
Main function; run automatically when called from command-line

removeSmallRegions()
A function to remove small region

saveRegions(*fileName=''*)
Save the region data into a new region file

class resynthesis.**ExecutorResynthesisExtensions**
Extensions to Executor to allow for specification rewriting and resynthesis. This class is not meant to be instantiated.

getCurrentStateAsLTL(*include_env=False*)
Return a boolean formula (as a string) capturing the current discrete state of the system (and, optionally, the environment as well)

resynthesizeFromNewSpecification(*spec_text*)

class specCompiler.**SpecCompiler**(*spec_filename=None*)

abortSynthesis()
Kill any running synthesis process.

compile()

loadSimpleSpec(*text=''*, *regionList=*[], *sensors=*[], *actuators=*[], *customs=*[], *adj=*[], *outputfile=''*)
Load a simple spec given by the arguments without reading from a spec file

For Slurp

region, sensors, actuators, customs are lists of strings representing props adj is a list of tuples [(region1,region2),...]

loadSpec(*spec_filename*)
Load the project object

ltlConjunctsFromBadLines(*to_highlight*, *useInitFlag*)

postprocessLTL(*text*, *sensorList*, *robotPropList*)

splitSpecIntoComponents(*env*, *sys*)

substituteMacros(*text*)
Replace any macros passed to us by the parser. In general, this is only necessary in cases where bitX propositions are needed, since the parser is not supposed to know about them.

unrealCores(*cmd*, *topo*, *badStatesLTL*, *conjuncts*, *deadlockFlag*)

unsatCores(*cmd*, *topo*, *badInit*, *conjuncts*, *maxDepth*, *numRegions*)

---

class strategy.**Domain**(*name*, *value_mapping=None*, *endianness=0*, *num_props=None*)
A Domain is a bit-vector abstraction, allowing a proposition to effectively have values other than just True and False.

Domain "x" consists of propositions "x_b0", "x_b1", "x_b2", ..., and the

value of the domain corresponds to the interpretation of these propositions as a binary string (following the order specified by *endianness*). If *value_mapping* is specified, the numeric value of the domain will be used as an index into this array, and the corresponding element (must be non-integer) will be returned when the proposition value is queried, instead of a number (likewise, when setting the value of the proposition, this mapping will be used in reverse).

num_props can be used to specify the size of the vector; if not

specified, this will be automatically calculated based on the size of the *value_mapping* array.

Define a domain: >>> animals = ["cat", "dog", "red-backed fairywren", "pseudoscorpion", "midshipman"] >>> d = Domain("favorite_animal", animals)

Notice that by using a domain we've reduced the number of props necessary >>> assert d.num_props == 3

Conversion can go in both directions: >>> for value in animals: ... p = d.valueToPropAssignments(value) ... assert value == d.propAssignmentsToValue(p)

B0_IS_LSB = 1

B0_IS_MSB = 0

getPropositions()
Returns a list of the names of the propositions that are covered by this domain.

numericValueToPropAssignments(*number*)
Convert an integer value into the corresponding dictionary [prop_name(str)->value(bool)] of propositions composing this domain

propAssignmentsToNumericValue(*prop_assignments*)
Convert a dictionary [prop_name(str)->value(bool)] of propositions composing this domain into an integer value.

propAssignmentsToValue(*prop_assignments*)
Return the value of this domain, based on a dictionary [prop_name(str)->value(bool)] of the values of the propositions composing this domain.

valueToPropAssignments(*value*)
Convert a value into the corresponding dictionary [prop_name(str)->value(bool)] of propositions composing this domain

class strategy.**State**(*parent*, *prop_assignments=None*)
A state, at its most basic, consists of a value assignment to propositions (represented as a dictionary {proposition name (string) -> proposition value}).

Additional metadata can be attached as necessary.

When created, a reference to the parent StateCollection needs to be be passed so that the state is aware of its evaluation context.

For example usage, see the documentation for StateCollection.

A Note About Multi-Valent Propositions:

Multivalent propositions (i.e. those whose value can span a Domain), are

handled fairly flexibly internally, but most of the dynamic translation is hidden from the user.

In general, read access to propositions will always be presented at the

highest-level possible. For example, when asking for the value of a state, multivalent propositions are presented instead of the underlying binary subpropositions (unless explicitly overridden by using the *expand_domains* flag provided by some functions). That said, if one wishes to query the value of a subproposition for some reason, its value will be calculated automatically.

In a similar vein, in order to minimize the worries of those using this

module, multivalent propositions can be written to– and are stored internally– in one of two ways: either as multiple binary assignments to the subpropositions of the domain, or a single value assignment to the domain proposition itself. This latter form is preferred, since it is simplest, and internal accounting is biased in this direction.

getAll(*expand_domains=False*)
Return a dictionary of assignments to all propositions for this state.

If *expand_domains* is True, return only the binary subpropositions for domains instead of the usual multivalent proposition.

getInputs(*expand_domains=False*)
Return a dictionary of assignments to input propositions for this state.

If *expand_domains* is True, return only the binary subpropositions for domains instead of the usual multi-valent proposition.

getLTLRepresentation(*mark_players=True*, *use_next=False*, *include_inputs=True*)
Returns an LTL formula representing this state.

If *mark_players* is True, input propositions are prepended with "e.", and output propositions are prepended with "s.".

If *use_next* is True, all propositions will be modified by a single "next()" operator. *include_env*, which defaults to True, determines whether to include input propositions in addition to output propositions.

getName()

getOutputs(*expand_domains=False*)
Return a dictionary of assignments to output propositions for this state.

If *expand_domains* is True, return only the binary subpropositions for domains instead of the usual multi-valent proposition.

getPropValue(*name*)
Return the value of the proposition *name* in this state.

(Note: *expand_domains* is not supported here because it would entail returning multiple values. Use getPropValues() for that.)

getPropValues(*names*, *expand_domains=False*)
Return a dictionary of assignments to the propositions in *names* for this state.

If *expand_domains* is True, return only the binary subpropositions for domains instead of the usual multi-valent proposition.

satisfies(*prop_assignments*)
Returns *True* iff the proposition settings in this state agree with all *prop_assignments*. Any unspecified propositions are treated as don't-cares.

setPropValue(*prop_name*, *prop_value*)
Sets the assignment of propositions *prop_name* to *prop_value* in this state. A lot of sanity checking is performed to ensure the name and value are both appropriate.

setPropValues(*prop_assignments*)
Update the assignments in this state according to *prop_assignments*.

Any existing assignments to propositions not mentioned in *prop_assignments* are untouched.

class strategy.**StateCollection**(**args*, ***kwds*)
StateCollection is a simple extension of list, to allow for keeping track of meta-information about states, such as which propositions are inputs and which are outputs, as well as domains. (These are not class properties of State because different StateCollections might have different settings.)

Create a new state collection: >>> states = StateCollection()

Define some basic true/false propositions: >>> states.addInputPropositions((“low_battery”,)) >>> states.addOutputPropositions((“hypothesize”, “experiment”, “give_up”))

Define some multi-valent propositions: >>> regions = [”kitchen”, “living”, “bedroom”] >>> animals = [”cat”, “dog”, “red-backed fairywren”, “pseudoscorpion”, “midshipman”] >>> states.addOutputPropositions([Domain(“region”, regions)]) >>> states.addInputPropositions([Domain(“nearby_animal”, animals, Domain.B0_IS_LSB)])

Create a new state: >>> test_assignment = {“region”: “bedroom”, “nearby_animal”: “midshipman”, ... “low_battery”: True, “hypothesize”: True, ... “experiment”:False, “give_up”:False} >>> s = states.addNewState(test_assignment) >>> assert s.satisfies(test_assignment)

You can also query and set the state values using low-level subpropositions: >>> s2 = states.addNewState(s.getAll(expand_domains=True)) >>> assert s2.satisfies(test_assignment)

LTL is available too! >>> s2.getLTLRepresentation() ‘!e.nearby_animal_b1 & !e.nearby_animal_b0 & e.nearby_animal_b2 & e.low_battery & s.region_b0 & !s.region_b1 & !s.give_up & !s.experiment & s.hypothesize’

addInputPropositions(*prop_list*)
Register the propositions in *prop_list* as input propositions. Each element of *prop_list* may be either a bare string, which is treated as the name of a binary proposition, or a Domain object, which indicates a multivalent proposition.

addNewState(*prop_assignments=None*, *goal_id=None*)
Create a new state with the assignment *prop_assignment* and goal ID *goal_id* and add it to the StateCollection.

Returns the new state.

addOutputPropositions(*prop_list*)
Register the propositions in *prop_list* as output propositions. Each element of *prop_list* may be either a bare string, which is treated as the name of a binary proposition, or a Domain object, which indicates a multivalent proposition.

clearPropositionsAndDomains()
Remove all propositions and domain definitions.

clearStates()
Remove all states.

expandDomainsInPropAssignment(*prop_assignments*)
Replace all domain propositions in the dictionary with their subpropositions

getDomainByName(*name*)
Returns the Domain object with name *name*.

If no such Domain is found, returns None.

getDomainOfProposition(*prop_name*)
Returns the Domain object for which proposition *prop_name* is a subproposition.

If no such Domain is found, returns None.

getPropositions(*expand_domains=False*)
Return a list of all known proposition names.

class strategy.**Strategy**
A Strategy object encodes a discrete strategy, which gives a system move in response to an environment move (or the reverse, in the case of a counterstrategy).

Only subclasses of Strategy should be used.

configurePropositions(*input_propositions*, *output_propositions*)
Set the input and output propositions for this strategy.

input_propositions and *output_propositions* must both be lists, consisting of any combination of strings (i.e. binary proposition names) and strategy.Domain objects (for multivalent propositions).

All existing definitions will be cleared.

This must be done before creating any states.

exportAsDotFile(*filename*, *starting_states=None*)
Output an explicit-state strategy to a .dot file of name *filename*. (For use with GraphViz.)

findTransitionableStates(*prop_assignments*, *from_state=None*)
Return a list of states that can be reached from *from_state* and satisfy *prop_assignments*. If *from_state* is omitted, the strategy's current state will be used.

iterateOverStates()
Returns an iterator over all known states.

loadFromFile(*filename*)
Load a strategy from a file.

searchForOneState(*prop_assignments*, *state_list=None*)
Iterate through all known states (or a subset specified in *state_list*) and return the first one that matches *prop_assignments*.

Returns None if no such state is found.

searchForStates(*prop_assignments*, *state_list=None*)
Returns an iterator for the subset of all known states (or a subset specified in *state_list*) that satisfy *prop_assignments*.

strategy.**TestLoadAndDump**(*spec_filename*)

strategy.**createStrategyFromFile**(*filename*, *input_propositions*, *output_propositions*)
High-level method for loading a strategy of any type from file.

Takes a filename and lists of input and output propositions. Returns a fully-loaded instance of a Strategy subclass.

class fsa.**FSAStrategy**
An automaton object is a collection of state objects along with information about the current state of the automaton when being executed.

findTransitionableStates(*prop_assignments*, *from_state=None*)
Return a list of states that can be reached from *from_state* and satisfy *prop_assignments*. If *from_state* is omitted, the strategy's current state will be used.

searchForStates(*prop_assignments*, *state_list=None*)
Returns an iterator for the subset of all known states (or a subset specified in *state_list*) that satisfy *prop_assignments*.

class bdd.**BDDStrategy**

BDDToPropAssignment(*bdd*, *var_names*)

BDDToState(*bdd*)

BDDToStates(*bdd*)

findTransitionableStates(*prop_assignments*, *from_state=None*)
Return a list of states that can be reached from *from_state* and satisfy *prop_assignments*. If *from_state* is omitted, the strategy's current state will be used.

getAllVariableBDDs(*use_next=False*)

getAllVariableNames(*use_next=False*)

getBDDFromJx(*jx*)

getJxFromBDD(*bdd*)

prime(*bdd*)

printStrategy()
Dump the minterm of the strategy BDD. For debugging only.

propAssignmentToBDD(*prop_assignments*, *use_next=False*)
Create a BDD that represents the given *binary* proposition assignments (expressed as a dictionary from prop_name[str]->prop_val[bool]). If *use_next* is True, all variables will be primed.

satAll(*bdd*, *var_names*)

satOne(*bdd*, *var_names*)

searchForStates(*prop_assignments*, *state_list=None*)
Returns an iterator for the subset of all known states (or a subset specified in *state_list*) that satisfy *prop_assignments*.

stateListToBDD(*state_list*, *use_next=False*)

stateToBDD(*state*, *use_next=False*)
Create a BDD that represents the given state. If *use_next* is True, all variables will be primed.

unprime(*bdd*)

CHAPTER 3

Handlers

3.1 Shared

3.1.1 dummySensor.py - Dummy Sensor Handler

Displays a silly little window for faking sensor values by clicking on buttons.

class `handlers.share.dummySensor.`**`sensorHandler`**(*proj*, *shared_data*)

`buttonPress`(*button_name*, *init_value*, *initial=False*)
Return a boolean value corresponding to the state of the sensor with name `sensor_name` If such a sensor does not exist, returns `None`

button_name (string): Name of the sensor whose state is interested init_value (bool): The initial state of the sensor (default=False)

`regionBit`(*name*, *init_region*, *bit_num*, *initial=False*)
Return the value of bit #bit_num in the bit-vector encoding of the currently selected region

name (string): Unique identifier for region sensor (default="target") init_region (region): Name of the sensor whose state is interested bit_num (int): The index of the bit to return

3.1.2 dummyActuator.py - Dummy Actuator Handler

Does nothing more than print the actuator name and state; for testing purposes.

class `handlers.share.dummyActuator.`**`actuatorHandler`**(*proj*, *shared_data*)

`setActuator`(*name*, *actuatorVal*, *initial*)
Pretends to set actuator of name `name` to be in state `val` (bool).

name (string): Name of the actuator

3.2 Pose

3.2.1 basicSimPose.py - 2D Pose provider for basicSimulator

class handlers.pose.basicSimPose.**poseHandler**(*proj*, *shared_data*)

getPose(*cached=False*)
Returns the most recent (x,y,theta) reading from basic simulator

3.2.2 NullPose.py - Pose Handler for single region without Vicon

class handlers.pose.NullPose.**poseHandler**(*proj*, *shared_data*, *initial_region*)

getPose(*cached=False*)

setPose(*x*, *y*, *theta*)

class handlers.pose.rosPose.**poseHandler**(*proj*, *shared_data*, *modelName='pr2'*)

getPose(*cached=False*)

3.2.3 viconPose.py - Pose Handler for Vicon System

class handlers.pose.viconPose.**poseHandler**(*proj*, *shared_data*, *host*, *port*, *x_VICON_name*, *y_VICON_name*, *theta_VICON_name*)

getPose(*cached=False*)

3.3 Drive

3.3.1 differentialDrive.py - Differential Drive Handler

Converts a desired global velocity vector into translational and rotational rates for a differential-drive robot, using feedback linearization.

class handlers.drive.differentialDrive.**driveHandler**(*proj*, *shared_data*, *d=0.6*)

setVelocity(*x*, *y*, *theta=0*)

3.3.2 holonomicDrive.py - Ideal Holonomic Point Drive Handler

Passes velocity requests directly through. Used for ideal holonomic point robots.

class handlers.drive.holonomicDrive.**driveHandler**(*proj*, *shared_data*, *multiplier*, *maxspeed*)

setVelocity(*x*, *y*, *theta=0*)

3.4 Motion Control

3.4.1 heatController.py - Potential Field Region-to-Region Motion Control

Uses the heat-controller to take a current position, current region, and destination region and return a global velocity vector that will help us get there

class handlers.motionControl.heatController.**motionControlHandler**(*proj*, *shared_data*)

get_controller(*current*, *next*, *last*, *cache={}*)
Wrapper for the controller factory, with caching.

gotoRegion(*current_reg*, *next_reg*, *last=False*)
If `last` is true, we will move to the center of the region.

Returns `True` if we are outside the supposed `current_reg`

3.4.2 vectorController.py - Vector Addition Motion Controller

Uses the vector field algorithm developed by Stephen R. Lindemann to calculate a global velocity vector to take the robot from the current region to the next region, through a specified exit face.

class handlers.motionControl.vectorController.**motionControlHandler**(*proj*, *shared_data*)

gotoRegion(*current_reg*, *next_reg*, *last=False*)
If `last` is True, we will move to the center of the destination region.

Returns `True` if we've reached the destination region.

3.4.3 RRTController.py - Rapidly-Exploring Random Trees Motion Controller

Uses Rapidly-exploring Random Tree Algorithm to generate paths given the starting position and the goal point.

class handlers.motionControl.RRTController.**motionControlHandler**(*proj*, *shared_data*, *robot_type*, *max_angle_goal*, *max_angle_overlap*, *plotting*)

buildTree(*p*, *theta*, *regionPoly*, *nextRegionPoly*, *q_gBundle*, *face_normal*, *last=False*)
This function builds the RRT tree. p : x,y position of the robot theta : current orientation of the robot regionPoly : current region polygon nextRegionPoly : next region polygon q_gBundle : coordinates of q_goals that the robot can reach face_normal : the normal vector of each face corresponding to each goal point in q_gBundle

createRegionPolygon(*region*, *hole=None*)
This function takes in the region points and make it a Polygon.

data_gen()

generateNewNode(*V*, *V_theta*, *E*, *Other*, *regionPoly*, *stuck*, *append_after_latest_node=False*)
Generate a new node on the current tree matrix V : the node matrix V_theta : the orientation matrix E : the tree matrix (or edge matrix) Other : the matrix containing the velocity and angular velocity(omega) information regionPoly: the polygon of current region stuck : count on the number of times failed to

generate new node append_after_latest_node : append new nodes to the latest node (True only if the previous node addition is successful)

getNode(*p*, *V*, *E*, *last=False*)
This function calculates the velocity for the robot with RRT. The inputs are (given in order):

p = the current x-y position of the robot

E = edges of the tree (2 x No. of nodes on the tree) V = points of the tree (2 x No. of vertices) last = True, if the current region is the last region

= False, if the current region is NOT the last region

getVelocity(*p*, *V*, *E*, *last=False*)
This function calculates the velocity for the robot with RRT. The inputs are (given in order):

p = the current x-y position of the robot E = edges of the tree (2 x No. of nodes on the tree) V = points of the tree (2 x No. of vertices) last = True, if the current region is the last region

= False, if the current region is NOT the last region

gotoRegion(*current_reg*, *next_reg*, *last=False*)
If `last` is True, we will move to the center of the destination region. Returns `True` if we've reached the destination region.

jplot()

orientation_bound(*theta*)
make sure the returned angle is between 0 to 2*pi

plotMap(*mappedRegions*)
Plotting regions and obstacles with matplotlib.pyplot

number: figure number (see on top)

plotPoly(*c*, *string*, *w=1*)
Plot polygons inside the boundary c = polygon to be plotted with matlabplot string = string that specify color w = width of the line plotting

3.5 Robot-specific handlers

TODO

CHAPTER 4

Configuration and Data Files

4.1 Definitions

- A :
- A `robot_file` describes the available functionalities of a given class of robot (i.e., sensor and actuator propositions), as well as the motion control strategy that should be used for driving the robot from region to region (e.g., potential field + differential-drive feedback linearization).
- A `spec_file` contains a specification, written in Structured English, which describes how the robot should behave.
- An `aut_file` is generated automatically from a `spec_file` and contains an automaton whose execution will cause the robot to satisfy the original specification (under environmental assumptions).
- An *Experiment Configuration File*

4.2 Example

To perform an experiment, the following steps should serve as a reasonable guideline:

1. Select a robot and import it into `specEditor`

4.3 Contents

4.3.1 Experiment Configuration File

TODO: Fixme

sd ads

ads whoah

CHAPTER 5

To-Do List

CHAPTER 6

Coding Conventions

Indentation is 4 space characters

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*