# Sembler Design Rule Checker Overview



Figure 1 Sembler Design Rule Checker workflow from DXF ingestion to output generation.

The goal of the Sembler Design Rule Checker (DRC) is to ascertain if a design submitted to the Sembler Rapid Microfluidics platform can be aggregated with other designs and fabricated. The DRC takes a given DXF design file, checks it against the design constraints of the aggregation and fabrication process, and reports any constraint violations found.

## Constraint Violation Types

Three types of constraint violations exist in the DRC. The first type of violation occurs when the geometry presented in the DXF design file cannot be parsed or converted into the geometry necessary for design rule checking. These violations include the existence of open or self-intersecting polylines, the use of unsupported DXF entities such as splines, incorrect layers or layer names, and an undefined device border. This type of violation is referred to in the user’s guide and PDF summary as a ‘purity’ error. In the codebase, it is referred to as a ‘purity’ error.

The second type of violation occurs when the design contains one or more elements that makes design aggregation or fabrication impossible. Unsupported SU8 layers or design elements that exist outside the device border are some examples of this type of violation. The user’s guide and PDF summary refers to these as ‘rules violations’ while the codebase refers to theses as ‘hard’ errors.

The last type of violations occurs when the design contains one or more elements that may result in the device not being fabricated correctly or the device not functioning as intended. However, the device design may still be aggregated with other designs and there is nothing that makes fabrication impossible. Examples of these violations include having too small of a pitch between elements on the same layer or between an element and the border. The user’s guide and PDF summary refers to these as ‘guidelines violations’ while the codebase refers to theses as ‘soft’ errors.

This document will use the terms ‘purity’, ‘hard’, and ‘soft’ when referring to the different types of DRC violations as these terms correspond to the nomenclature in the codebase.

## Workflow

The Sembler DRC workflow begins with a DXF design file that describes a device with features on up to three SU8 layers and a metal layer. The geometry in the design file is parsed by the DRC twice, the first time to check for purity errors and the second to populate the internal representation of the design in a DesignDict object. The design rules are then checked against the design in the DesignDict object. Any Failures (hard and soft errors) in design rule checking, if any exist, are reported in a both a PDF summary file and a DXF summary file. The PDF summary contains description of all errors (if any) on each layer of the DXF design and a diagram of where the error exists on each layer. The DXF summary creates a new layer for each error, where each error includes multiple violations of the same design constraint, and highlights the location of each design rule violation with a witness mark.

## Code Organization and Packages

The DRC codebases consists of three top level files (sembler.sh, sembler.py, sql\_interactions.py), and the packages in the table below.

|  |  |
| --- | --- |
| **Package** | **Summary** |
| checker | Describes the design and the design rules |
| config | Configurations for constants and boilerplate text for PDF summary |
| dxfparser | Validates the design and converts geometries from Shapely format to DXF format and vis versa |
| intermediate | Describes rules violations and witness marks |
| tests | Unit tests |
| golden\_test | System level tests |

# Sembler Top Level Files

Three files exist in the top level of the project. The main python file (sembler.py) contains the entry point to the design rule checker (DRC), and contains the code that ties together the DXF file ingestion, design validation, and output generation. A wrapper shell executable (sember.sh) handles any failures of the python process or being unable to read or write to the input and output directories. Finally, the interactions with the SQL database on the server is handled by the functions in the file sql\_interactions.py.

## Sembler (sembler.py)

This file contains the main execution code for the design rule checker. It handles argument parsing, building the internal representation of the design from the input DXF, finding design rule violations, and writing the output to both a DXF summary and a PDF summary.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| purityCheck(dwg) | Perform design purity check against the DXF design file. Set purity errors field of the resultJSON dictionary, and return a dictionary of purity failures found (if any). |
| populateDD(designDict, layers) | Converts an ingested DXF into a DesignDict. The DXF input ingest twice because a large amount of additional data is computed during construction of a design dictionary such as buffered geometry, spatial index, and intent of features. |
| printFailures(purityFailures, designDict, resultJSON, checkplots, outDir) | Generates the summary PDF output by the DRC process and populates the errors field of the resultJSON dictionary, including how many hard and soft errors were found by the DRC, and adds the path to the summary PDF |
| checkRules(purityFailures, designDict, checkPlots, resultJSON, filename, filePath, outDir) | Given an ingested DXF in the form of a DesignDict, check it against all design rules and report the number of hard and soft errors. |
| generateCheckPlots(designDict, resultJSON, filename, outDir) | Produces plots of the locations of hard and soft errors in a DesignDict, which are the included in the summary PDF |
| loadDXF(filename) | Parse a DXF file from a path |
| parseArguments() | Pulls command line arguments into more canonical forms and generates the help messages for when the wrong number of arguments are provided, or –help is submitted. |
| areArgumentsValid(filename, inDir, outDir, projectID) | Makes sure the file, input and output directories exist. Returns True if arguments are valid, False otherwise |
| configureLogger(filename, outputDirectory) | Sets up the logging file, and logger levels |
| writeJSON(resultJSON, jsonFilename) | Produces the resulting JSON file associated with running the DRC. |
| createFabricationOutput(filename, inputDirectory, outputDirectory) | Generates a GDS file for fabrication. Alters the design by adding support posts to each layer they affect and adding all ports to the layer SU8\_1. |
| runDRC(filename, inputDirectory, outputDirectory, resultJSON) | Runs the DRC from input ingestion to output summary creation. |
|  |  |
| main() | Runs the DRC end to end on an input file |

## SQL Interactions (sql\_interactions.py)

This file sets up the connection to the SQL database on the external server, and contains the functions for setting project IDs, getting project names, and getting and setting the status of projects on the server. The username and password for logging into the database are set here. Database interactions are ignored if the python library MySQLdb does not exist. On startup, this file sets the database to be accessed, and the database cursor.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| setProjectID(pid) | Sets the project ID to the integer pid |
| getName(projectID, cursor) | Fetches the name of the project (user submitted DXF file name) from the database. If no name is fetched, none returns. |
| getStatus(projectID, cursor) | Gets the current status of the design rule checker of the project with ID projectID. |
| setStatus(status, projectID, cursor, db) | Updates the design rule checker status of the project projectID in the database db. |

# Internal Representation of Designs

## Configurations

All configuration files for the DRC are in the config package. Names and constants related to parsing the DXF design are listed in DXFconfig.py, including valid and required layer names, valid DXF entities, and the thresholds for heuristics used in the DRC. RulesConfig.py specifies the name, description, violation type (hard or soft), ID, and threshold value of all design constraints checked by the DRC. Boilerplate text from the summary PDF output is described in PurityConfig.py for purity errors and SummaryConfig.py for all other errors. The configuration files are summarized in the following table.

|  |  |
| --- | --- |
| **Configuration File** | **Summary** |
| DRCConfig.py | DRC version number, floating point equality fuzzing constant |
| DXFConfig.py | Required layer names, valid DXF entities, arc discretization constants, threshold for closed polylines and tiny entities |
| PurityConfig.py | Summary text for unreadable DXF files |
| RulesConfig.py | Rule name, heading, and description for summary output files, threshold values for checking rules |
| SummaryConfig.py | Standard text for different types of summary files (pass, pass with conditions, failed, unreadable) |
| Translations.py | Maps layer names in DXFConfig to layer numbers in RulesConfig |

## Layer (Layer.py)

A Layer represents a logical layer of the microfluidic device such as the metal wafer or a layer of SU8. Each layer in the DXF design file is mapped to a Layer object. Objects on the layer are contained in both a dictionary and a spatial index (rectangular tree). Valid layer names are listed in DXFConfig.py and in the table below.

|  |  |
| --- | --- |
| **Layer Name** | **Description** |
| METAL | Metal traces on the glass |
| SU8\_1 | 20um tall SU8 layer |
| SU8\_2 | 30um tall SU8 layer |
| SU8\_3 | 50um tall SU8 layer |
| VPORT | Layer defining location of all ports in device |
| POSTS | Layer defining location of all support posts in device |
| ALIGNMENT | Layer containing all alignment marks, which will be projected onto the METAL and SU8\_1 layers during rule checking |
| BORDER | Layer containing border defining outermost boundary of the device |

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| id | Instance variable, integer that corresponds to the layer name as described in Translations.py |
| objCount | The number of objects in the layer |
| name | The name of the layer |
| index | Spatial index |
| objs | Dictionary of objects on the layer |
| add(obj) | Adds an object to both the dictionary and the spatial index |
| addCollection(objs) | Adds a group of objects to the dictionary but the not the spatial index. Incomplete and will raise an exception if used. |

## Design Object (DesignObject.py)

The Object base class represents the objects in a microfluidic design such as channels and ports. It is an abstract class that should never be instantiated directly. The subclasses Port, Border, Feature, AlignmentMark or Post should be used instead.. All dimensions for objects are in µm.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| count | Class variable, Total count of all design objects that have been instantiated |
| strOfType() | Returns the type of the object (Port, Border, Feature, AlignmentMark) as a string |
| bounds() | Returns the bounding box of the object as a list of coordinate tuples |
| shape | Instance variable, represents shape of the object as a Shapely polygon |
| type | Instance variable, type of the object (Port, Border, Feature, AlignmentMark) |

### Port

A Port (subclass of Object) object represents a location on the device where a biopsy punch will be used to cut through all objects present in those locations on SU8 layers. Ports may either hold fluid or be relief cuts for attaching electrodes to the metal layer and are always circular. Instantiating a port requires a point (float, tuple) as the center of the port and a radius (float). A Port is assigned a purpose as different design rules apply to fluid bearing ports and relief cuts. The information is not contained in the DXF file, so the method setPurpose(metalIndex)assigns the purpose depending on if the Port intersects with any feature on the metal layer.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| center | Instance variable, xy-cooridates of the center of the port as a tuple |
| radius | Instance variable, radius of the port |
| purpose | Instance variable, declares a port to hold fluid or act as relief cut for attaching electrodes |
| count | Class variable, Total count of all Port objects that have been instantiated in the design |
| setPurpose(metalIndex) | Sets the purpose of the port by checking if it intersects with a feature on the metal layer or not. If it intersects with a feature on the metal layer, then it is assumed to be a relief cut to allow electrodes to be attached. Otherwise, it is assumed that the port will contain fluid. |

### Border

A Border (subclass of Object) represents the outermost border of the device. This is used to calculate the dimensions of the device.

### Feature

A Feature (subclass of Object) represents a channel in the SU8 layers or a trace in the metal layer of the device. Initializing a Feature requires a polygon that describes the shape of the feature.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| diameter | Instance variable, smaller of the width or height of the feature. Used to calculate if the feature can be supported by lower layers |
| purpose | Instance variable, sets purpose of the feature as either CHANNEL or SUPPORT |
| count | Class variable, total count of Feature objects that have been instantiated in the design |

### Post

A Post (subclass of Feature) represents a support post designed to hold up channels that would otherwise be too wide and prone to collapse. A Post has a defined depth, the uppermost layer with a feature that the post intersects. The default depth for a Post is the metal layer.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| depth | The layer which the top of the support reaches or the layer that the support post is ‘holding up’. |

### Alignment Mark

Alignment marks exist to ensure the proper alignment of the SU8 layers with the underlying metal during fabrication as stipulated in the original DXF. As such, they exist in multiple layers (Metal and SU81) simultaneously. Certain rules apply to alignment marks that do not apply to other features in either Metal or SU8-1 layers, and therefore, AlignmentMark is a subclass of Object instead of Feature.

The shape of each AlignmentMark is identical and is defined in the \_\_init\_\_(center, ident) method. An AlignmentMark consists of a square cross 500µm wide with crossbars 100µm wide centered at the point center. An exception is raised if a center is not defined.

## Design Dictionary (DesignDict.py)

The DesignDict class wraps the objects in the parsed DXF design into a single logical item and handles the workflow from ingestion from the DXF to running rules against the ingested design and reporting the rules violations. The main data structure inside a DesignDict is a list of Layer objects with the following IDs, in order, where each Layer corresponds to a layer of the DXF design file: : L0\_METAL, L1\_SU8, L2\_SU8, L3\_SU8, SUPPORT\_POST, VERTICAL\_PORT. A list of the variables and methods of the DesignDict class are shown below.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| dimm | Length (x) and width (y) of design in a tuple |
| objects | Dictionary of all Objects in the design |
| ports | Layer containing all Port objects in the design |
| alignments | Dictionary of all AlignmentMark objects in the design |
| layers | List of Layer objects, in the following order L0\_METAL, L1\_SU8, L2\_SU8, L3\_SU8, SUPPORT\_POST, VERTICAL\_PORT |
| border | Polygon that defines the device border |
| violations | Summary of all design rule violations present in the device design |
| addToLayer(obj, layer) | Adds a DesignObject to the specified layer of the DesignDict. Objects that occur inside ports are not added, with the exception of bondpads on the metal layer. |
| getLayer(id) | Accesses the specified layer of the device contained in the DesignDict |
| addPort(port) | Adds a Port object port to the VERTICAL\_PORT layer |
| getPorts() | Returns the collection of ports from the VERTICAL\_PORT layer |
| populateSize(runJson) | Computes the size of the design using the information from the border layer. Uses the function roundedSub(max, min) (rounds to the smaller of floor(max-min) or ceil(max-min)) for subtraction. The dimensions calculated here are used to check rules about minimum device size. The size of the design is written to the JSON file runJson |
| addAlignment(align) | Adds an AlignmentMark object to both L0\_METAL and L1\_SU8 layers. |
| addPost(post) | Adds a Post to the SUPPORT\_POST layer |
| getAlignments() | Returns a dictionary of AlignmentMark objects present in the design |
| violationCount() | Compute and returns the number of violations (rules and guidelines) associated with the design dictionary. Returns 0, even if the design dictionary contains errors, if this is run before rules are checked. |
| getPortIndex() | Deprecated code. Port index is now tracked by the Port class |
| setBorder(b) | Sets the Border object b as the border for the design dictionary |
| check(rid) | Checks the design in the design dictionary against the rule associated with the index rid. |
| checkAllThreaded() | Checks the design in the design dictionary against all the rules in the rules dictionary. This implementation checks each rule in its own thread. |
| checkAllSingleThread() | Checks the design in the design dictionary against all the rules in the rules dictionary. This implementation checks all rules in a single thread. |
| checkAll() | Wrapper for one of checkAllThreaded() or checkAllSingleThreaded() |
| getViolationCounts() | Counts the number of violations in the design dictionary and returns a tuple of hard violations and soft violations. If this is run before checking the rules, the result will be (0,0) even if there are potential violations in the design dictionary. |
| getObjCounts() | Returns a dictionary mapping the name of each layer to the number of objects in that layer. |

Rule checks on a DesignDict can either be run in a single thread or each rule checked in its own thread. The class CheckThread, subclass of Thread handles creating a new thread for each rule to be checked. The class is initialized with a ID threadID, the rule to be checked (rule), and the DesignDict to check the rule against (dd).

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| threadID | ID of the thread |
| rule | Rule the thread checks |
| dd | the DesignDict to check the rule against |
| run() | Checks the rule against the design in the DesignDict |

# DXF Ingestion

The DRC process begins with ingestion of the design in DXF format and parsing it. A DRC failure will be thrown if the initial file is not a DXF file, and the process will end. The input file is first scanned for purity errors that would produce geometries that cannot be verified by the design rules. A design without purity errors will be converted to Shapely polygons and passed on for rules checking. The library dxfgrabber is used to parse the DXF file.

## DXF Validation (dxfvalidator.py)

Purity errors fall under several distinct categories: invalid design border, missing or invalid DXF layers, or invalid geometry. Invalid geometry can be further broken down into unsupported DXF entities, DXF polylines that self-intersect, or open DXF polylines. DXF entities that exist in inserts are also checked for purity errors. In addition, all DXF entities smaller than a certain threshold as defined in DxfConfig.py (‘tiny’ entities) are moved to their own layer for the user to review as these are likely to be artifacts and will hinder design rule checking.

All files with purity violations are returned as having failed design rule checking in the final output reports.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| checkDrawingPurity(drawing) | Checks drawing for all possible purity errors |

### BadInsert

This class is a simple container to associate objects with the entities they contain. This is used for recording purity violations that occur inside inserts.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| insert | DXF insert containing a purity error |
| block | DXF block (container of objects) the insert references |

### Invalid Border

The Border layer of the DXF file contains the entity used to calculate the size of the device boundary, and as such, must be convertible into a single rectangle using the Shapely library. The Border layer must contains one and only one entity, and that the entity must be a polyline. All other results will generate a purity violation.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| \_checkForInvalidBoundary(drawing) | Checks that the drawing only has one entity that is a polyline on the Border layer. |

### Invalid DXF Layers

All the layers listed as valid layers in DxfConfig.py must exist in the drawing, otherwise a DesignDict mapping features to layers will not be able to be generated.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| \_checkForValidLayers(drawing) | Checks that all required layers are present in the drawing |
| \_isOnValidLayer(entity) | Deprecated. Checks if the entity is on a required layer. |

### Self-Intersecting Polylines

All polylines in the drawing (including the polylines in any inserts) must not self-intersect. A polyline is considered self-intersecting if an attempt to convert the polyline into a Shapely polygon returns a ‘self-intersection’ error. Such polylines cannot be converted into Shapely geometry necessary for rules checking.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| \_checkForSelfIntersectingPolylines  (drawing, layers) | Checks all entities (including those inside inserts) on all layers of a drawing for self-intersecting polylines |
| \_checkPolylineSelfIntersecting  (polyline) | Checks if a specific polyline has one or more self-intersections. |
| \_checkInsertForIntersectingPolylines(insert, drawing) | Checks all entities (including inserts) inside an insert for self-intersecting polylines |

### Open Polylines

All polylines in the drawing (including the polylines in any inserts) must be closed or they will not be convertible into Shapely geometry for rules checking.. The process will attempt to close open polylines that have a gap of less than the threshold value CLOSED\_POLYLINE\_THRESHOLD defined in DxfConfigy.py (default = 3 µm). If the polyline cannot be closed, it will be reported as an open polyline and generate a purity violation.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| \_checkForOpenPolylines(drawing, layers) | Checks all entities (including those inside inserts) on all layers layers of a DXF file drawing for open polylines |
| \_isPolylineOpen(entity, attemptClose=True) | Checks if a specific polyline entity is open and if the gap is small enough (as defined in DxfConfig.py) to attempt to close it |
| \_closePolyline(polyline) | Closes an open polyline polyline with a gap of smaller than the threshold |
| \_checkInsertForOpenPolylines  (insert, drawing) | Checks all entities (including inserts) inside an insert insert in a DXF drawing drawing for open polylines |

### Invalid DXF Entities

All entities in the DXF drawing (including entities in inserts and blocks used by inserts) must be of a DXF type listed as valid in DxfConfig.py. The only valid entity types are: LINE, CIRCLE, ARC, POLYLINE, LWPOLYLINE, and INSERT.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| \_checkFoInvalidEntityTypes(drawing, layers) | Checks all entities (including those inside inserts) on all layers layers of a DXF file drawing for invalid DXF entities. |
| \_checkInsertForInvalidEntities(insert, drawing, validEntityTypes) | Checks all entities (including inserts) inside an insert insert in a DXF drawing drawing for invalid DXF entities. The list of valid entity types is included in validEntityTypes |

### Tiny Entities

All entities with a size below a given heuristic threshold (defined in DxfConfig.py), depending on entity type are moved to an error layer in the output DXF summary for the user to review.

The heuristic measurement and threshold for each entity type is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Entity Type** | **Measurement** | **Threshold** | **Value** |
| POLYLINE/LWPOLYLINE | Perimeter | TINY\_POLYLINE\_THRESHOLD | 1 |
| CIRCLE | Radius | TINY\_RADIUS\_THRESHOLD | 1 |
| LINE | Line length | TINY\_LINE\_LENGTH\_THRESHOLD | 1 |
| ARC | Arc length | TINY\_LINE\_LENGTH\_THRESHOLD | 1 |

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| findTinyEntities(drawing) | Finds all tiny entities (including those in inserts) in the DXF drawing drawing and moves them to a separate layer for user review |
| isEntityTiny(entity) | Compares an entity against thresholds set in DxfConfig.py to check if it is a tiny entity |
| \_checkInsertforTinyEntities(insert, drawing) | Checks all entities (including inserts) inside an insert insert in a DXF drawing drawing for tiny entities. |
| \_moveTinyInsertToLayer(insert, layerName) | Moves tiny entities present inside an insert to the specified layer. |
| \_\_calculatePerimeter(polyline) | Calculate the perimeter of a given polyline |

## Geometry Ingestion (dxfparser.py)

The DXF drawing is converted by layer into Shapely objects after checks for purity errors are completed. The functions in dxfparser.py are used to ingest the DXF design file in preparation for conversion into Shapely geometry by listing all the entities that exist on each layer of the DXF drawing. The functions involved in transformations are used in extracting entities from inserts.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| separateLayers(dwg, layerNames) | Separates a DXF drawing dwg by layer name and returns the entities of each specified layer in a dictionary layerNames. Insert entities inside the drawing are exploded into the individual entities they describe. |
| separateStandardLayers(dwg) | Separates the standard layers in the drawing dwg as defined in DxfConfig.py |
| separateFabricationLayers(dwg) | Separates the layers in the drawing dwg needed to generate the fabrication output in GDS |
| translateEntity(entity, offset) | Translates an entity entity IN-PLACE by the given offset (dx, dy). |
| rotateEntity(entity, theta, origin) | Rotates an entity entity IN-PLACE by the given angle theta about the given origin origin (x,y). |
| scaleEntity(entity, scale) | Scales an entity entity IN-PLACE by the given scale factor scale (sx, sy, sz). Currently only capable of scaling polylines and circles. Circles can only be scaled uniformly in X and Y. Non-uniform scales would create ellipses. |
| extractInsertEntities(dwg, insert, layerName) | Takes a DXF insert insert inside the drawing dwg and explodes it into the corresponding entities it describes if the insert is on layer layerName. Performs all scale, rotation, and translations as appropriate. |
| parseInsertEntities(dwg, layerName) | Searches through a drawing dwg for inserts on layer layerName and explodes them into the entities they describe. |

### Test Plot Functions (dxfplotter.py)

The functions in this file are used to generate test plots of DXF designs in python to verify that the file was read in correctly. This includes functions to plot DXF entities including circles and polylines. Functions present in this file are not used outside of test functions.

## Geometry (dxf2shapely.py)

DXF entities are converted into Shapely objects (Point, Polygon) on which design rules are run. The functions in the file dxf2shapely.py handles all the conversions between valid DXF entities and Shapely geometry. During this conversion process, all DXF arcs and curved portions of DXF polylines are discretized into line segments using the thresholds for discretization set in DxfConfig.py. Positive and negative space will be identified in nested DXF polylines and handled accordingly as interior holes in Shapely Polygon objects. The geometry for each DXF layer is encapsulated in a Shapely Multipolygon or as a list of Shapely Polygon objects.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| \_convertBulge(start, end, bulge) | Converts an AutoCAD arc of two points (start, end) and a bulge value bulge to a center point, radius, start and end angle. The formula uses http://www.afralisp.net/archive/lisp/Bulges1.htm as a guide. |
| \_angleBetweenPoints(p1, p2) | Returns the angle of the line between two points p1 and p2 made to the X-axis in radians. |
| \_polar(pt, angle, distance) | Returns the point at a specified angle angle (in radians) and distance distance from a starting point pt. This function mirrors the functionality of AutoLISPs polar function |
| \_discretizeArc(arc, nPoints=50) | Takes an arc arc specified as (center, radius, startAngle, endAngle) and discretizes it into nPoints line segments |
| \_linspace(a, b, n=100) | Creates a list of n linearly spaced values between a and b |
| polyline2polygon(polyline, offsets=(0, 0)) | Creates a shapely Polygon object from a polyline polyline offset by offset and discretizes any arcs found in the polyline. |
| polyline2linearRing(polyline, offsets=(0, 0)) | Deprecated. Creates a shapely linear ring object from a polyline polyline offset by offset. Currently does not convert arcs. |
| polyline2lineString(polyline, offsets=(0, 0)) | Deprecated. Creates a shapely line string object from a polyline polyline offset by offset. Currently does not convert arcs. |
| circle2shapelyCircle(circle, offsets=(0, 0)) | Takes a DXF circle object circle offset by offset and discretizes it to create a Shapely Polygon. |
| \_circle2DRCCircle(circle, offsets=(0, 0)) | Converts a DXF circle object circle offset by offset to an object readable by the rule checker. Only used when converting Port objects |
| convertLayer(layer) | Takes in a list of DXF entities from layer layer and converts them to the correct Shapely object |
| setify(collection) | Discards all duplicate Shapely elements in the given collection collection. |
| quadTreeSetify(collection) | Discards all duplicate Shapely elements in the given collection collection using a quad tree as a container |
| bufferLayer(layer, distance) | Takes all Shapely objects on a given layer layer and buffers them by a fixed value distance. |
| findShapesInHoles(shapes, layer) | Find all Shapely polygons that exist inside holes in a list of Shapely polygons shapes in a given Shapely Multipolygon layer |
| findContainedShapes(shapes, layer) | Finds all Shapely polygons that are fully contained within the polygons shapes in a given Shapely Multipolygon layer |
| nestedUnion(layer) | Unions together all shapes in a given Shapely Multipolygon layer, taking into account nesting. Shapes with nested geometry will be treated correctly as positive or negative space. |
| calculatePolarity(shapelyLayer, layerName=None) | Converts all DXF entities in a list shapelyLayer on a layer layerName to Shapely shapes, taking into account nesting. Shapes with nested geometry will be treated correctly as positive or negative space. |

# Design Rule Checking

Design rule/constraint checking begins after the DXF design has been completely converted in Shapely objects. The Shapely libraries includes a number of convenient built-in functions for geometric computations, and as such, are used extensively in the design rule checks. The configurations and specifications for each design rule are stored in the file RulesConfig.py. The class Rule represents the abstract structure for a design constraint while individual rules are represented as subclasses of Rule.

## Rule (Rule.py)

The Rule abstract class contains the representation of the rules to be checked by the DRC. This class wraps the naming / display information for the output summary PDF and DXF, as well as the logic for checking the design against the rule. Each constraint in the DRC is represented as a subclass of Rule. The definition for each rule in RulesConfig.py contains the information assigned to the instance variables id, desc, expl, layers, dxfLayer, ctype, and threshold. The rule IDs listed in RulesConfig.py were originally based on spreadsheet numbering of the rules, so the IDs are not in numerical order.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| id | Instance variable. Numerical ID of the rule |
| desc | Instance variable. One sentence description of the rules. Used in generating the output summary PDF of the results |
| expl | Instance variable. Explanation of the details behind the rule violation. Used in generating the output summary PDF of the results |
| layers | Instance variable. Which layers of the input design the rule affects. |
| dxfLayer | Instance variable. The name of the DXF layer that errors are drawn on in the output summary DXF. |
| ctype | Instance variable. Where the rule is a hard failure (cannot be fabricated) or a soft guideline violation (can be fabricated but there is a chance the design will not function as intended) |
| threshold | Instance variable, float. The limits of success or failure for the rule. |
| count | Class variable. Tracks the total number of rules that have been instantiated. |
| dict | Class variable. A dictionary where each rule is mapped to its ID. A Rule is added to the dictionary when it is first instantiated. |
| check(designDict) | Check the for compliance with the rule. If there are violations, add them to the design dictionary. Subclasses are expected to override this method. |

## Design Rules

Two types of design rules exist in the DRC, hard rules and soft rules. Violation of hard rules means that the physical constraints for the design aggregation and fabrication process have not be fulfilled, and the Sembler fabrication process cannot continue. In this case, the DRC will return a result of ‘failed’. Hard rules include rules on border size, alignment marks, and supported layers. Soft rules represent best design practices for the Sembler fabrication process, and may be violated at the user’s risk and discretion. In this case, the DRC will return a result of ‘passed with conditions’, and the Sembler fabrication process will not guarantee successful fabrication and function of the device. For example, collapsed features may occur if the maximum width of a feature on a layer is exceeded or features may not be properly defined in the PDMS if the minimum width of a feature is not met.

The design rules have been organized into groups based on what device features or aspects of device features they concern. Design rules concerning the border, the alignment marks, and supported layers are hard rules while design rules concerning feature size and pitch, ports, and support posts are soft rules. Helper functions for rules checking are listed at the end of the section.

### Witness Marks (Witness.py)

Each Rule is associated with a Witness object. Witness objects are visual feedback of violated rules and highlight issues on the error layers in the DXF summary file for the users. The type of Witness object generated for each design is specified in the check(dd) method for each rule, and not in a configuration file. Four types of Witness objects exist: Region, LineSegment, Point, and PointRadius. Regions are used to highlight unsupported geometry, geometry that is outside the boundary, and errors in the number of alignment marks in the output DXF file. Line segments are specified by the functions in the file NearbySegment.py and highlights a line between two points in the output DXF. Line segments are used to show errors in pitch between features if the features are too close together or too far apart. Point-radius Witness objects are rendered as circles in the output DXF file and are used to highlight errors in minimum and maximum feature width. Point Witness objects are only used to highlight the self-intersection purity error.

The Witness object for each design rule is listed in the rule description tables. In addition several of the purity errors do not generate Witness objects. In those cases, the lack of witness marks is reported in the logger.

### Border

The main constraints for the border are the size and shape. The border must be a rectangle with edges of 1, 2, 3, or 4 cm, or the size of 2.5cm by 7.5 cm microscope slide. In addition, all features on the layers L0\_METAL, L1\_SU8, L2\_SU8 and L3\_SU8 must be fully contained inside the border. All design rules concerning the border are hard rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| MaxDieDimm.py | 4 | Hard | Checks that the larger dimension of the border is less than or equal to 4cm or is equal to 7.5 cm | Region |
| MinDieDimm.py | 3 | Hard | Checks that the smallest die dimension is no smaller than the threshold set in RulesConfig.py (default: 1cm) | Region |
| BorderSize.py | 39 | Hard | Checks that unless the border is the size of a microscope slide (2.5cm x 7.5cm), the size of each dimension of the border is 1.0cm, 2.0cm, 3.0cm, or 4.0cm | Line Segment |
| RectBorder.py | 40 | Hard | Checks that the border is a rectangle | Line Segment |
| L1SU8InBounds.py | 34 | Hard | Checks that all objects on layer L1\_SU8 are inside the die boundary. | Region |
| L2SU8InBounds.py | 41 | Hard | Checks that all objects on layer L2\_SU8 are inside the die boundary. | Region |
| L3SU8InBounds.py | 42 | Hard | Checks that all objects on layer L3\_SU8 are inside the die boundary. | Region |
| MetalInBounds.py | 33 | Hard | Checks that all objects on layer L0\_METAL are inside the die boundary. | Region |

### Alignment Marks

Design rules about the alignment marks are concerned with the number of the alignment marks, the pitch between alignment marks, and the pitch between the alignment mark and the border. All design rules concerning alignment marks are hard rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| AlignmentBorderPitch.py | 31 | Hard | Checks that alignment marks are a distance of no smaller than the threshold set in RulesConfig.py from the border as measured from the nearest edge of the alignment mark. (default: 750µm) | Line Segment |
| AlignmentCount.py | 32 | Hard | Checks that designs contain at least 3 alignment marks if the largest dimension of the device is 2cm or less, or at least 6 if the largest dimension of the device is larger than 2cm | Region |
| AlignmentPitch.py | 30 | Hard | Check that the alignment marks present are a distance of less than the threshold defined in RulesConfig.py apart, and that no more than 2 alignment marks are co-linear. | Line Segment |

### Supported Layers

All features on higher layers (L2\_SU8 and L3\_SU8) must be completely supported by layers below them. A layer is supported if all the features on that layer can be completely contained by the features on the layer below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| SupportedSU82.py | 17 | Hard | Checks that all features present on layer L2\_SU8 are supported underneath by features on layer L1\_SU8. | Region |
| SupportedSU83.py | 22 | Hard | Checks that all features present on layer L3\_SU8 are supported underneath by features on layer L2\_SU8. | Region |

### Feature Size

Both minimum and maximum feature sizes are determined by which layer of SU8 the features in questions reach. Features wider than the maximum may collapse while features smaller than the minimum may not be properly patterned from PDMS. Layer L1\_SU8 (with a height of 20µm) may have features between 10 µm and 80µm. Layer L2\_SU8 (with a height of 50µm combined) may have features between 25 µm and 200µm. Layer L3\_SU8 (with a height of 100µm combined) may have features between 50 µm and 400µm. Features on the metal layer have a minimum feature width of 10µm but no maximum. All design rules concerning feature size are soft rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| MaxSU81Width.py | 10 | Soft | Checks that the widest feature on layer L1\_SU8 is no larger than the threshold specified in RulesConfig.py (default: 80µm) | Point-radius |
| MaxSU82Width.py | 14 | Soft | Checks that the widest feature on layer L2\_SU8 is no larger than the threshold specified in RulesConfig.py (default: 200µm) | Point-radius |
| MaxSU83Width.py | 19 | Soft | Checks that the widest feature on layer L3\_SU8 is no larger than the threshold specified in RulesConfig.py (default: 400µm) | Point-radius |
| MinMetalWidth.py | 5 | Soft | Checks that the narrowest feature on layer L0\_METAL is no smaller than the threshold specified in RulesConfig.py (default:10 µm) | Point-radius |
| MinSU81Width.py | 9 | Soft | Checks that the narrowest feature on layer L1\_SU8 is no smaller than the threshold specified in RulesConfig.py (default:10 µm) | Point-radius |
| MinSu82Width.py | 13 | Soft | Checks that the narrowest feature on layer L2\_SU8 is no smaller than the threshold specified in RulesConfig.py (default:25 µm) | Point-radius |
| MinSU83Width.py | 18 | Soft | Checks that the narrowest feature on layer L3\_SU8is no smaller than the threshold specified in RulesConfig.py (default:50 µm) | Point-radius |

### Feature Pitch

Feature pitch refers to the distance edge-to-edge between two features on the same layer or between a feature and the border. Features too close together may not be clearly defined in the PDMS while features too close to the border may hinder bonding between the PDMS and the glass and cause leaks. All design rules concerning feature pitch are soft rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| MetalEdgePitch.py | 6 | Soft | Checks the distance edge-to-edge between the features on the L0\_METAL layer is no less than the threshold specified in RulesConfig.py (default: 10µm) | Line Segment |
| SU81EdgePitch.py | 35 | Soft | Checks the distance edge-to-edge between the features on the L1\_SU8L layer and the die boundary is no less than the threshold specified in RulesConfig.py (default: 750µm) | Line Segment |
| SU82EdgePitch.py | 43 | Soft | Checks the distance edge-to-edge between the features on the L2\_SU8 layer and the die boundary is no less than the threshold specified in RulesConfig.py (default: 750µm) | Line Segment |
| SU83EdgePitch.py | 44 | Soft | Checks the distance edge-to-edge between the features on the L3\_SU8L layer and the die boundary is no less than the threshold specified in RulesConfig.py (default: 750µm) | Line Segment |
| SU81Pitch.py | 12 | Soft | Checks the distance edge-to-edge between the features on the L1\_SU8 layer is no less than the threshold specified in RulesConfig.py (default: 50µm) | Line Segment |
| SU82Pitch.py | 16 | Soft | Checks the distance edge-to-edge between the features on the L2\_SU8 layer is no less than the threshold specified in RulesConfig.py (default: 50µm) | Line Segment |
| SU83Pitch.py | 21 | Soft | Checks the distance edge-to-edge between the features on the METAL layer is no less than the threshold specified in RulesConfig.py (default: 50µm) | Line Segment |

### Ports

The allowed radii of ports is determined by the standard port punches Sembler uses during fabrication. The minimum distance edge-to-edge between ports is necessary so the physical act of punch a port will not disturb adjacent ports. Ports holding fluid should be a minimum distance from the device border to allow for a good bonding area between the PDMS and the glass, but that is not required for ports meant as relief cuts for attaching electrodes. As all features on SU8 layers must be supported by the features on layer L1\_SU8, checking the pitch between ports and features on L1\_SU8 is sufficient for finding the minimum pitch between ports and features on any layer in the device. While alignment marks are projected on to layer L1\_SU8, they are classified as non-fluid holding features and are ignored during the check for the pitch between features and ports. All design rules concerning ports are

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| PortDimm.py | 23 | Soft | Checks that the diameter of each port is 1mm, 2mm, or 3mm. Ports not of those diameters will not be punched. | Point-radius |
| PortPitch.py | 24 | Soft | Checks the distance edge-to-edge between the ports is no less than the threshold specified in RulesConfig.py (default: 50µm) | Line Segment |
| FluidPortEdgePitch.py | 25 | Soft | Checks the distance edge-to-edge between the ports that carry fluid and the die boundary is no less than the threshold specified in RulesConfig.py (default: 2000µm) | Line Segment |
| RelPortEdgePitch.py | 26 | Soft | Checks the distance edge-to-edge between the ports that are relief cuts for electrodes and the die boundary is no less than the threshold specified in RulesConfig.py (default: 0µm) | Line Segment |
| SU81PortPitch.py | 27 | Soft | Checks the distance edge-to-edge between ports and features on layer L1\_SU8 is no less than the threshold specified in RulesConfig.py (default: 1000µm) | Line Segment |

### Bondpads

Bondpads are locations on the device where electrodes will be attached. In the design, they appear as overlaps between features on the layer L0\_METAL and the ports on the layer PORTS. A Witness is not drawn when a bondpad is smaller than the minimum width. All design rules concerning bondpads are soft rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| BondpadPitch.py | 38 | Soft | Checks the distance edge-to-edge between bondpads is no less than the threshold specified in RulesConfig.py (default: 2000µm) | Line Segment |
| MetalBondPitch.py | 8 | Soft | Checks the distance center-to-center between features on the METAL layer and bondpads is no less than the threshold specified in RulesConfig.py (default: 500µm) | Line Segment |
| MinBondpadWidth.py | 36 | Soft | Checks the narrowest bondpads are no smaller than the threshold specified in RulesConfig.py (default: 100 µm) No witnesses are drawn for this. | N/A |

### Support Posts

Support posts are used when a feature on a layer needs to be wider than the maximum width. The support posts help the feature to not collapse due to a low aspect ratio. The minimum post size and maximum distance edge-to-edge between posts are dependent which layer the support posts reach. All design rules concerning support posts are soft rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Rule ID** | **Type** | **Summary** | **Witness** |
| MaxPostPitch.py | 44,45,46 | Soft | All three rules are combined in this file. Checks that the distance edge-to-edge between support posts on each layer is less than the thresholds set in RulesConfig.py. (defaults: L1\_SU8: 50 µm, L2\_SU8: 200 µm, L3\_SU8: 400 µm) | Region |
| MinPostDimmSU81.py | 11 | Soft | Checks that the diameter of the support posts reaching layer L1\_SU8 is greater than the thresholds set in RulesConfig.py. (default: 20 µm) | Point-radius |
| MinPostDimmSU82.py | 15 | Soft | Checks that the diameter of the support posts reaching layer L2\_SU8 is greater than the thresholds set in RulesConfig.py. (default: 50 µm) | Point-radius |
| MinPostDimmSU83.py | 20 | Soft | Checks that the diameter of the support posts reaching layer L3\_SU8 is greater than the thresholds set in RulesConfig.py. (default: 100 µm) | Point-radius |

## Helper Functions

While simple rules can be checked by direct application of built-in Shapely functions, more complex rules require helper functions for geometry manipulations. Helper functions exist for calculating the minimum and maximum feature width, and the edge-to-edge distance between two features.

### LayerBorderBuffer.py

The functions contained in LayerBorderBuffer.py are used calculate the distance between all features on a layer and the border of the design. The rules SU81EdgePitch.py, SU82EdgePitch.py and SU83EdgePitch.py are checked using these functions.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| checkThresh(obj1, obj2, thresh) | Measure distance between 2 Objects obj1 and obj2 using a built-in Shapely function for measuring distances between polygons. Returns true if the distance is less than the threshold thresh. |
| buffered(rule, dd, layerID, thresh) | For every object in the layer layerID, check that the distance edge-to-edge between the object and the border referenced in the DesignDict dd is greater than the threshold thresh specified by the Rule rule. Otherwise generate a violation and add it to the DesignDict dd. |

### LayerMinDist.py

The functions contained in LayerMinDist.py are used calculate the distance between all pairs of features on a given layer. The rules MetalEdgePitch.py, SU81Pitch.py, SU82Pitch.py and SU83Pitch.py are checked using these functions.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| checkThresh(obj1, obj2, thresh) | Measure distance between 2 Objects obj1 and obj2 using a built-in Shapely function for measuring distances between polygons. Returns true if the distance is less than the threshold thresh. |
| minDistInLayer(rule, dd, layerID, thresh) | For each object in layer layerID, find all candidate objects within thresh distance of the object. For the object and each candidate object, check that the object does not contain the candidate object and vice versa, and that the distance between them is greater than thresh distance. Otherwise generate a violation and add it to the Design Dictionary dd. |

### LayerMinLinewidth.py

The functions in LayerMinLineWidth.py are used in calculating the minimum feature width in a layer. Rules that use the function minLWInLayer to check geometry are MinSU1Width.py, MinSU82Width.py, MinSU83Width.py, MinMetalWidth.py while the rule MinBondpadWidth.py uses the function checkObs.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| check\_points(polygon, minwidth) | For each internal or external point in the Shapely polygon polygon, generate a circle with radius minwidth centered on that point. Take the difference between the generated circle and the polygon. If the remaining object is not a polygon, generate a witness mark at that point. |
| interiorRectangl(p0, p1, minWidth) | Generates a line segment from Points p0, p1, and makes an parallel line segment minWidth away to the right. Returns a rectangle defined by those two line segments. |
| exteriorRectangle(p0, p1, minWidth) | Generates a line segment from Points p0, p1, and makes an parallel line segment minWidth away to the left. Returns a rectangle defined by those two line segments. |
| addPolygons(ring, interior, minWidth, collection) | For every consecutive pair of coordinates in the list of points ring, make a line segment. If that line segment is longer than the minimum distance constant posDelta, make a rectangle to the interior or exterior a distance minWidth away, and append it to the list collection. |
| minLWInLayer(rule, dd, layerID, threshold) | For every object in Layer layerID, first check the minimum width of the object’s shape at every vertex and keep a list of witness marks generated. Then generate interior and exterior rectangles for every line segment making up the object at a distance threshold. For each rectangle, shrink it by a scale of -1 using the built-in Shapely function buffer. The Rule rule is violated if the original object intersects with the shrunken rectangle. If there are any witnesses or violating rectangles, a violation is added to the DesignDict dd. |
| checkObs(rule, dd, objs, threshold) | For every object in Layer layerID, generate interior and exterior rectangles for every line segment making up the object a distance threshold. For each rectangle, shrink it by a scale of -1 using the building in shapely function buffer. The Rule rule is violated if the original object intersects with the shrunken rectangle. If there are any witnesses or violating rectangles, a violation is added to the DesignDict dd. Used only to check for minimum width of bondpads (MinBondpadWidth.py) |

### NearLineSegment.py

The functions in the file NearLineSegment.py defines the line segment used as a witness mark between two features or ports. The center of a port is always chosen as an endpoint of the line segment.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| pointToPolygon(point, polygon) | Takes a Shapely Point point and a Shapely Polygon polygon. Converts the exterior of the polygon into a Shapely LinearRing, and projects the point onto the linear ring, getting the distance. Move along the ring to get the point of interested, p. Return the pair of points point, p as the ends of a line segment. |
| polygonToPolygon(poly1, poly2) | Take the exterior of both Shapely Polygons poly1 and poly2 as linear rings. For each point in the exterior of poly1, project onto poly2, and get the distance, selecting the point on poly1 is the closest to poly2. Reduce to the function pointToPolygon |
| portToFeature(port, feat) | Returns the endpoints of a line segment between the center of the Port port and the exterior of the Feature feat. |
| portToPort(port1, port2) | Returns the centers of Port port1 and Port port2 as a tuple |
| featureToFeature(feat1, feat2) | Returns the endpoints of the shortest line segment between Feature feat1 and Feature feat2 |
| featureToBorder(feat, border) | Returns the endpoints of the shortest line segment between Feature feat and Border border |
| portToBorder(port, border) | Returns the endpoints of the shortest line segment between the center of Port port and Border border |
| alignmentToBorder(am, border) | Returns the endpoints of the shortest line segment between the center of AlignmentMark am and Border border |
| alignmentToAlignment(am1, am2 | Returns the centers of AlignmentMark am1 and AlignmentMark am2 as a tuple |

### PostDepth.py

The functions in the file PostDepth.py are used to determine the layer that any given support post is used to support. The rules defined in MaxPostPitch.py, MinPostDimmSU81.py, MinPostDimmSU82.py, and MinPostDimmSU83.py use these functions to determine the minimum pitch between posts and minimum post diameters.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| deepestLayer(post, designDict) | For the given Post post, check it against all objects in layers L3\_SU8, L2\_SU8, and L1\_SU8, with layers in that order, in the DesignDict designDict. If an object in a layer completely contains the post, return the depth of the post as that layer. If a feature overlaps a post but does not completely contain it, return an error. |
| fillInPosts(designDict) | For each post in the layer POSTS in the Design Dictionary designDict, find the deepest layer that the post reaches. |

### SamplingSize.py

The functions in the file SamplingSize.py are used to a) convert a Shapely polygon into a collection of triangles and sample random points from each triangle and b) find the maximum distance between the centroid of a polygon after subtracting any support posts it overlaps and a set of randomly sampled points from inside that polygon. These are used in calculating the maximum feature size in the file MaxLineWidth.py.

|  |  |
| --- | --- |
| Function | Summary |
| perPolygon(poly) | Converts Polygon poly into a collection of triangles using the built-in function triangulate from Shapely. Returns the number of random samples in each triangle (total of 1000), the collection of triangles, and a version of the polygon that has be prepared for further geometric operations (Shapely functions contains, covers, intersects) |
| prepPolygonSafe(poly) | Same as perPolygon, only that an exception is thrown if the Polygon poly cannot be triangulated. |
| prepPolygon(poly) | Attempts to triangulate the Polygon poly with perPolygon(poly), and uses prepPolygonSafe(poly) if it fails. |
| nRandomTriangleSamples(poly, triangles) | For each triangle in collection triangles, sample a number of points and see if they are inside the polygon poly. Returns the list of points sampled over all triangles. |
| nRandomBBoxSamples(poly, n=SAMPLES) | Takes n samples (default: 1000) over the bounding box of Polygon poly. Return the list of sampled points |
| nRandomSamples(poly) | If ratio of the area of the Polygon poly to the area of its bounding box is greater than 0.5, take random samples over the bounding box. Otherwise triangulate the polygon and take random samples over the triangles. |
| maxMinDistance(poly, dd) | Finds the maximum width of a Polygon poly in DesignDict dd, taking into account support posts and ports that overlap or otherwise intersect poly. Any support post completely contained by poly is subtracted from the interior of poly. The maximum distance from randomly sampled points from the interior of the construct to the centroid of the construct is returned, along with the point creating the distance. |
| visResult(poly, currentPoint, dist) | Test function. Not used in production code. |
| test(poly) | Test function. Not used in production code. |
| visPolyTriangle(poly) | Test function. Not used in production code. |

### MaxLineWidth.py

The function in this file calculates the maximum width of the features in a given layer of the design, taking into account the features that support the current layer but ignore structures that do not stop at the current layer. This function is used by rules defined the files MaxWidthSU81.py, MaxWidthSU82.py, and MaxWidthSU83.py.

|  |  |
| --- | --- |
| **Function** | **Summary** |
| checkMaxWidth(rule, dd, layerID, threshold) | Find the maximum width of the features on the layer layerID, including the features supporting the current layer but discounting the structures that reach above this layer. Widths greater than the threshold distance threshold will be reported as a violation in the DesignDict dd as a violation of Rule rule. The width is calculated using the function maxMinDistance from the file SamplingSize.py. |

### SupportCheck.py

The functions in the file SupportCheck.py are used to check that geometry in the upper SU8 layers L2\_SU8 and L3\_SU8 are fully supported by the layers beneath them. Used to check geometry in SupportedSU82.py and SupportedSU83.py

|  |  |
| --- | --- |
| **Function** | **Summary** |
| supported(below, above) | Takes two Objects, below and above, and checks if the shape of above is fully contained inside of or is equal to the shape of below. |
| checkSupport(rule, dd, belowLayer, aboveLayer) | For all Objects in the layer aboveLayer, check that they are fully contained by an object in the layer belowLayer, not counting support posts. Any violations will get reported in the DesignDict dd as a violation of Rule rule. |

## Violations (Violation.py)

Violation objects exist to record all the rules violations in the design. Each rule generates one Violation, and the Violation contains all the elements of the design that violate the rule. Each Violation also contains all the witness marks that need to be drawn in the output DXF to highlight the errors.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| count | Class variable. The total number of violations generated |
| id | Instance variable. ID to reference an individual violation |
| ruleID | Instance variable. The rule that the violation is associated with |
| name | Instance variable. The name of the violation for the ruleID as described in the RuleConfig.py |
| desc | Instance variable. The detailed description of the violation for the ruleID as described in RuleConfig |
| dxfLayer | Instance variable. The layer that the witness marks will be drawn on the summary dxf file |
| conflicting | Instance variable. The list of objects that caused the violation |
| witnesses | Instance variable. The list of |
| inserts | Instance variable. The list of inserts associated with the violation. |
| layer | Instance variable. The layer of the design that the violation took place on |
| ctype | Instance variable. The type of rule that was violated (Purity, Soft, Hard) |
| \_\_str\_\_() | Returns a string containing the rule ID, the description, and all objects that violated the rule. |
| ofRule(rule, entities, witnesses) | Generates a Violation for the Rule rule with a list of entities that violated the rule and the witness marks that highlight those entities. |

### Violation Summary (ViolationSummary.py)

The class ViolationSummary is an abstract representation of a set of rules violations and keeps counts of the number and type of violations as well as how many violations are on each layer. The information contained in this structure is used to generate a summary table in the output summary PDF.

|  |  |
| --- | --- |
| **Variables and Methods** | **Summary** |
| violations | Instance variable. List of all violations included in the summary |
| violationsByLayer | Instance variable. Dictionary of layers and the violations on each layer |
| purityViolationsByLayer | Instance variable. Dictionary of layers and the purity violations on each layer |
| hardViolationsByLayer | Instance variable. Dictionary of layers and the hard violations on each layer |
| softViolationsByLayer | Instance variable. Dictionary of layers and the soft violations on each layer |
| hardCount | Instance variable. Total number of hard violations across all layers |
| softCount | Instance variable. Total number of soft violations across all layers |
| purityCount | Instance variable. Total number of purity violations across all layers |
| sawHard | Instance variable. Boolean, true if a hard violation exists on any layer |
| sawSoft | Instance variable. Boolean, true if a soft violation exists on any layer |
| tex() | Converts information contained in the 3 dictionaries into a LaTeX table reporting the number of each type of violation per layer |
| add(violation) | Adds a Violation violation to the summary. Updates each dictionary and counter with the Violation. |
| ofList(vList) | Construct a ViolationSummary object from a list of violations vList |
| ofDict(vDict) | Construct a ViloationSummary from a dictionary vDict, assuming each violation ID maps to a single Violation. |

# Output Summary

When the DRC is complete, the user receives the result in both a PDF summary and a DXF file highlighting the locations of the rules violations, if any. Four different types of outputs summaries exist:

* Pass: no rules violations of any kind were found
* Pass with conditions: soft rules violations were found. Design aggregation and fabrication may continue but the user assumes the risk of having a non-functional device.
* Fail: hard rules violations were found, and these errors must be fixed for the design to be aggregated and fabricated.
* Unreadable: purity errors were found and the DRC could not continue.

The boilerplate text for each of the summary types is found in the SummaryConfig.py file.

## PDF Summary

The PDF summary output provides an overview of all violations, if any, found in the design. A table with the number and type of each rules violation is presented, along with summary diagrams that highlight the location of each violation. The DRC generates a LaTeX file, and then uses the program pdflatex to convert the LaTeX document into a PDF. The LateX file is created by the functions in TexOutput.py, and the summary diagrams (checkplots) are generated by the functions in Visualizations.py.

### TexOutput.py

The functions in TexOutput.py are used to generate a table from the ViolationSummary associated with the design, and to add the description, header, and other information found in RulesConfig.py for each rule violated by the design to the output summary.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| mangleFile(path) | Given the DXF file name path, produce the appropriate PDF and LaTeX output locations. |
| sectionString(sname) | Adds the given section sname to the LaTeX output file |
| subsectionString(sname) | Adds the given subsection sname to the LaTeX output file |
| ruleVString(sname) | Adds a rule violation section for the given rule sname to the LaTeX output file. |
| purityVString(sname) | Adds a purity check violation section for the given rule sname to the LaTeX output file |
| suggString(sname) | Adds a guideline violation section for the given rule sname to the LaTeX output file. |
| addImage(path) | Adds an image at location path to the LaTeX output file |
| checkplotString(paths) | Adds labeled subsections containing a checkplot where the label reflects the layer in the checkplot where all images are found in the input path paths. |
| violationDescString(violation) | Adds a section showing a violation violation of a rule. Takes an actual rule as input, and determines if a rule or suggestion was violated based on that rule. |
| countString(i) | Count the number of times a given rule was violated by a design. |
| violationString(violation, vcount) | Given a Violation violation and the number of locations vcount in the design that caused this, produces an entire rule section, including the rule's subheader, the long description of that rule, and text stating how many times the rule was violated. |
| generateViolatedRuleOutput(  vSummary) | Builds a summary table of rules / suggestions violated by layer from the ViolationSummary vsummary. |
| statusString(vSummary) | Generates the boilerplate for the summary table based on the type of violations present in the design from the ViolationSummary vsummary. |
| generateExplanatoryBP(inputDxf, status) | Generates the standard boilerplate text for every design regardless of rule violations. |
| generateOutputBP(asDXF, asPDF, outDXF, vSummary) | Generate the boilerplate text detailing the name of the output DXF file with errors highlighted if errors exist in the input design. |
| generateFailureType(vSummary) | Based on the violations present, return the type of PDF summary to be generated from the ViolationSummary vsummary. |
| outputPDF(violationSummary, checkplotPaths, dxfPath, outDir, vstring) | Produces a summary pdf of the DRCs evaluation of a given design dictionary. Takes a variety of input, including violations found by the DRC (purityViolations, violationList), paths to checkplots of the design (checkplotPaths), the path to the input DXF file (dxfPath) and an output directory to drop the pdf into (outdir). |

### Visualization.py

The functions in the file Visualizations.py are used to draw diagrams showing the location on each DXF layer of any rules violations. These images are saved into a temporary directory and added to the output summary PDF.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| makeAlignmentMark(cX, cY) | Produce the shapely representation of an alignment mark centered at cX, cY. |
| mangleFile(filename) | Returns the filename filename with no extension and stripped of characters that LaTeX may not parse. |
| commandString(inFile, outFile) | Produce the command string for generating PDF output from the input inFile to the output outFile from geojson using ogr2ogr |
| drawLayer(dd, layerID, outPath, drawPort = True) | Renders a layer (layerID) of the DesignDict (dd) as a PDF, stored at outPath. |
| allJsonLayers(dd, outPath) | Produce file names and geojson images representing each layer. |
| convertLayers(outDir, basePath, (drewMetal, drewPortAlignment, drewSU81, drewSU82, drewSU83)) | Convert all of the geojson images to PDFs representing the layers |
| drawAllLayers(dd, filename, outDir) | Given the DesignDict dd, render the design and the checkplots for the design, and convert them to PDF files in the directory outDir. |

## DXF Summary

The DXF output summary is produced by adding layers containing witness marks highlighting the rules violations to a copy of the original DXF design. The functions in the file dxfwriter.py are used to add error layers and render witness marks.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| generatePurityFailureOutput(filenameNoExt, filePath, outDir, purityFailures) | Creates the annotated DXF from a list of purity failures |
| generateDRCFailureOutput  (filenameNoExt, filePath, outDir, violations): | Creates the annotated DXF from a dictionary of violations. |
| \_\_renderWitness(witness, drawing, layerName) | Pick the function for rendering each type of witness (region, line segment, circle, or point) |
| \_\_renderPoint(point, drawing, layerName) | Renders a Point type witness |
| \_\_renderRegion(region, drawing, layerName) | Renders a Region type witness |
| \_\_renderLineSegment  (lineSegment, drawing, layerName) | Renders a Line Segment type witness |
| \_\_renderPointRadius  (pointRadius, drawing, layerName) | Renders a Circle type witness |
| \_parseBadInsert(insert, layer, dwg) | Takes an insert and moves all entities within it to the specified layer |
| \_getEntityFromDwg(dwg, handle) | Gets an entity matching the specified handle from a drawing |
| \_moveToLayer(entity, layer, dwg) | Moves an entity by handle to the specified layer |
| \_\_polygon2Polyline(dwg, coords, attrs) | Takes a list of coordinates and creates a polyline from it with the given attributes |
| \_\_convertPoint2Marker(point, drawing, layerName) | Draws a marker with a circle of 100um diameter and crosshairs of 120um centered on the coordinates of point on the DXF layer (layername) |
| \_\_convertPoints2Line(points, drawing, layerName) | Given two points in list points, draw a line between them in the DXF drawing (drawing) on the layer layerName. |
| \_\_convertPointRadius2Circle(pointRadius, drawing, layerName) | Given a point and a radius, draw the circle with that radius centered on the point on layer layerName in the DXF drawing drawing. |
| \_\_convertPolygon2Polyline  (dwg, shape, layerName) | Takes a shapely Polygon and converts it to a DXF polyline on the layer layername in the DXF drawing dwg. |
| \_createLayer(dwg, layerName, color=10) | Creates a layer in the specified drawing dwg with the specified color. This will not overwrite existing layers of the same name. |

## GDS Fabrication File

The functions in the file gdswriter.py use the Python package gdspy to convert the Shapely objects in the DesignDict into a gds file for fabrication.

|  |  |
| --- | --- |
| **Functions** | **Summary** |
| generateFabricationOutput(filenameNoExt, outDir, shapelyLayers) | Maps each layer in DxfConfig to a GDS layer number. For each layer and list of objects in the dictionary shapleyLayers, convert all DXF Circles and Shapley objects into GDS objects, and write the resulting output to the output path (outdir/filenameNoExt) |
| \_\_convertDXFCircle2gds  (topCell, circle, layerNumber) | Converts DXF circles into GDS Round objects. |
| \_\_convertShapely2gds  (topCell, shape, layerNumber) | Converts Shapely objects into GDS polygons. |

# Code Manifest

Sembler

* sembler.sh
* sembler.py
* sql\_interactions.py
* checker
  + rules
    - AlignmentBorderPitch.py
    - AlignmentCount.py
    - AlignmentPitch.py
    - BondpadPitch.py
    - BorderSize.py
    - FluidPortEdgePitch.py
    - L1SU8InBounds.py
    - L2SU8InBounds.py
    - L3SU8InBounds.py
    - LayerBorderBuffer.py
    - LayerMinDist.py
    - LayerMinLinewidth.py
    - MaxDieDimm.py
    - MaxLineWidth.py
    - MaxPostPitch.py
    - MaxSU81Width.py
    - MaxSU82Width.py
    - MaxSU83Width.py
    - MetalBondPitch.py
    - MetalEdgePitch.py
    - MetalInBounds.py
    - MinBondpadWidth.py
    - MinDieDimm.py
    - MinMetalWidth.py
    - MinPostDimmSU81.py
    - MinPostDimmSU82.py
    - MinPostDimmSU83.py
    - MinSU81Width.py
    - MinSU82Width.py
    - MinSU83Width.py
    - NearLineSegment.py
    - PortDimm.py
    - PortPitch.py
    - PostDepth.py
    - RectBorder.py
    - RelPortEdgePitch.py
    - SamplingSize.py
    - SU81EdgePitch.py
    - SU81Pitch.py
    - SU82EdgePitch.py
    - SU82Pitch.py
    - SU83EdgePitch.py
    - SU83Pitch.py
    - SupportCheck.py
    - SupportedSU82.py
    - SupportedSU83.py
  + DesignDict.py
  + DesignObject.py
  + Layer.py
  + Rule.py
  + TexOutput.py
  + Visualization.py
* config
  + DRCConfig.py
  + DxfConfig.py
  + PurityConfig.py
  + RuleConfig.py
  + SummaryConfig.py
  + Translations.py
* dxfparser
  + dxf2shapely.py
  + dxfparser.py
  + dxfplotter.py
  + dxfvalidator.py
  + dxfwriter.py
  + gdswriter.py
* intermediate
  + Violation.py
  + ViolationSummary.py
  + Witness.py

# Installation

## Python Version

Sembler requires Python 2.7.x.

## External Libraries

Sembler requires the following external Python libraries to be installed:

* dxfgrabber-0.7.4 (located at fs5-Projects/MAMBO/Design Rule Checker/dependencies)
* Rtree 0.8.2
* Shapely 1.5.16
* ezdxf 0.6.4
* decartes 1.0.2
* matplotlib 1.5.1
* gdspy 0.7.1

## Setting proxies

In order to access the python package repositories, it may be necessary to set the following proxies in a posix based windows shell:

export draper\_proxy\_url='proxyvip.draper.com'

export draper\_proxy\_port=3128

export dproxy=$draper\_proxy\_url:$draper\_proxy\_port

export HTTPS=$dproxy

export HTTP=$dproxy

export https\_proxy=$dproxy

export http\_proxy=$dproxy

## Running

To run locally: python sembler.py <filename> <indir> <outdir>

* filename: name of the design file
* indir: directory the design file is located in
* outdir: directory output summary PDF and DXF files should be saved to

If the user does not have LaTeX installed, TexOutput.py needs to be edited to remove the check for the existence of the program pdflatex. In this case, a .tex file will be generated for the summary instead of a PDF.