Modeling Geometry with Assemblies in SysML [*]

March 25, 2016

Abstract

In this module a preliminary concept implementation is provided for the possible introduction of a novel kind of 3D diagram in SysML. Such "Assembly" Diagram in used to specify an operable description of the 3D geometry of a system part.

Contents

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. March 25, 2016

1 Introduction

1.1 bbbbbbbb

2 Implementation

2.1 Diagram initialization

Uniform cell sizing A cuboidal 3-complex is generated by the script below, where the cells have uniform dimension on each coordinate direction.

⟨ Diagram initialization 1 ⟩ ≡

```
""" Diagram initialization """
def assemblyDiagramInit(shape):
   print "\n shape =",shape
   # shape must be 3D, i.e. a python array with 3 indices
   assert len(shape) == 3
   diagram = larCuboids(shape)
   return diagram
◇
```

Macro never referenced.

Non-uniform cell sizing The parameter `quoteList` is used here to generate the new vertices of the `diagram`, previously generated with uniform spacing between the cell vertices in every coordinate direction. Each `pattern` in `quoteList` is a list of positive numbers, each corresponding to the size of the corresponding ”coordinate stripe”.

⟨ Diagram initialization (non-uniform sizing) 2a ⟩ ≡

```
""" Diagram initialization """
def assemblyDiagramInit (shape):
   def assemblyDiagram (quoteList):
      print "\n shape =",shape
      # shape and quoteList must be 3D, i.e. a python array with 3 indices
      assert (len(shape) == 3) and (len(quoteList) == 3)
      coordList = [list(cumsum([0]+pattern)) for pattern in quoteList]
      verts = CART(coordList)
      _,CV = larCuboids(shape)
```

```
        return verts,CV
    return assemblyDiagram
◇
```

Macro referenced in 7a.

Diagram scaling to cuboid of given size The `size` parameter is the array of lateral dimensions to which to scale the `diagram` parameter. `size` must be an array of 3 numbers; `diagram` is a LAR model

⟨Diagram scaling to sized cuboid 2b⟩ ≡

```
""" Diagram scaling to given size """
def unitDiagram(diagram, size=[1,1,1]):
    V,CV = diagram
    print "\n shape =",shape
    # size must be a python array with 3 numbers
    assert (len(size) == 3) and (AND(AA(ISNUM)(size)) == True)
    V_ = array(V) / AA(float)(max(V))
    V = (V_ * size).tolist()
    diagram = V,CV
    return diagram
◇
```

Macro referenced in 7a.

2.2 Diagram segmentation

Boundary cells ($3D \to 2D$) computation The computations of boundary cells is executed by calling the `boundaryCells` from the `larcc` module.

⟨Boundary cells ($3D \to 2D$) computation 2c⟩ ≡

```
def lar2boundaryFaces(CV,FV):
    """ Boundary cells computation """
    return boundaryCells(CV,FV)
◇
```

Macro referenced in 7a.

Interior partitions ($3D \to 2D$) computation The indices of the boundary 2-cells are returned in `boundarychain2D`, and subtracted from the set $\{0, 1, \ldots, |E| - 1\}$ in order to return the indices of the `interiorCells`.

⟨Interior partitions ($3D \to 2D$) computation 3a⟩ ≡

```
def lar2InteriorFaces(CV,FV):
   """ Boundary cells computation """
   boundarychain2D = boundaryCells(CV,FV)
   totalChain2D = range(len(FV))
   interiorCells = set(totalChain2D).difference(boundarychain2D)
   return interiorCells
◇
```

Macro referenced in 7a.

2.3 Subdiagram mapping

The aim of this section is to allow for separate development of subdiagrams of a geometric diagram. When satisfied with the current design situation, the developer may map a whole diagram into a single 3D cell of the upper-level diagram — in the following called the *master* diagram. Of course, such nesting may happen several times within a (father) master, producing a hierarchical decomposition (of any depth) of the geometry diagrams.

Task decomposition The procedure to map a diagram to a sub diagram is described below in a top-down manner, decomposing the task into an ordered set of subtasks.

The `diagram2cell` functions below works as follows. Its job is to map the LAR model `diagram` (semantically a 3-array of cuboidal blocks) onto the 3D-cell of the `master` LAR model (another 3-array of cuboidal blocks), indexed by the integer `cell` parameter. In few words: mapping `diagram` onto the given `cell` of `master`.

First, the matrix `mat`of this 3D-window to 3D-viewport transformation is computed, by invoking `diagram2cellMatrix`. Then, the (mat) transformation is applied to `vertices`. Then both such LAR models are passed as parameters of the `vertexSieve` function, that returns a single vertex list `V`, two (reindexed) lists `CV1` and `CV2`, and the number `n12` of common vertices.

We can look at their common incidence matrix as shown in Figure ??.

⟨ Subdiagram to diagram mapping 3b ⟩ ≡

```
⟨ 3D window to viewport transformation 5 ⟩

def diagram2cell(diagram,master,cell):
   mat = diagram2cellMatrix(diagram)(master,cell)
   diagram =larApply(mat)(diagram)
   (V1,CV1),(V2,CV2) = master,diagram
   n1,n2 = len(V1), len(V2)

   # identification of common vertices
   V, CV1, CV2, n12 = vertexSieve(master,diagram)
   commonRange = range(n1-n12, n1)
```

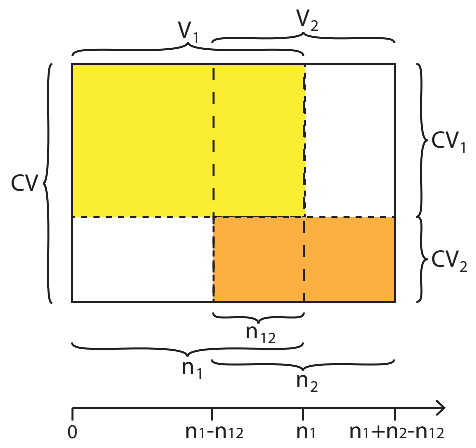

Figure 1: Structure of the characteristic matrix $M(CV)$ afre the merge of two LAR models, and identification of the common vertices.

```
newRange = range(n1,n1-n12+n2)

# addition of incident vertices into the adjacents of theCell
def checkInclusion(V,theCell,newRange):
   theVerts = [V[v] for v in theCell]
   theMin, theMax = min(theVerts), max(theVerts)
   theCell += [v for v in newRange if (
      theMin[0] <= V[v][0] and theMin[1] <= V[v][1] and theMin[2] <= V[v][2]
      and
      V[v][0] <= theMax[0] and V[v][1] <= theMax[1] and V[v][2] <= theMax[2]
      )]
   return theCell

# addition of new vertices into the adjacents of cell c
CV1 = [checkInclusion(V,c,newRange)
      if set(c).intersection(commonRange) != set() else c
       for c in CV1]

# masterBoundaryFaces = boundaryOfChain(CV,FV)([cell])
# diagramBoundaryFaces = lar2boundaryFaces(CV,FV)
CV = [c for k,c in enumerate(CV1) if k != cell] + CV2

master = V, CV
return master
```
◇

Macro referenced in 7a.

3D window to viewport transformation

⟨ 3D window to viewport transformation 5 ⟩ ≡

```
""" 3D window to viewport transformation """
def diagram2cellMatrix(diagram):
   def diagramToCellMatrix0(master,cell):
      wdw = min(diagram[0]) + max(diagram[0])        # window3D
      cV = [master[0][v] for v in master[1][cell]]
      vpt = min(cV) + max(cV)                       # viewport3D
      print "\n window3D =",wdw
      print "\n viewport3D =",vpt

      mat = zeros((4,4))
      mat[0,0] = (vpt[3]-vpt[0])/(wdw[3]-wdw[0])
      mat[0,3] = vpt[0] - mat[0,0]*wdw[0]
      mat[1,1] = (vpt[4]-vpt[1])/(wdw[4]-wdw[1])
      mat[1,3] = vpt[1] - mat[1,1]*wdw[1]
      mat[2,2] = (vpt[5]-vpt[2])/(wdw[5]-wdw[2])
      mat[2,3] = vpt[2] - mat[2,2]*wdw[2]
      mat[3,3] = 1
      print "\n mat =",mat
      return mat
   return diagramToCellMatrix0
```
◇

Macro referenced in 3b.

3 Topological consistency

When a 3D diagram is generated as a Cartesian product of 1D complexes, it is relatively easy to compute its cells of any dimension. For this purpose, see the the module `largrid` and/or the function `gridSkeletons(shape)`, that returns the list of skeletons generated by the cellular complex of a given `shape`.

Two different strategies may be used to guarantee the correctness of topology after local refinements, that provide a replacement of single cells with subdivided complexes. Such two strategies are discussed and developed in the next two subsections.

3.1 Decomposition of the whole space

As already coped with in module `larcc`, the facets, i.e. the $(d-1)$-faces, of a cellular d-complex may be easily computed using the product of the sparse characteristic matrix M_d times its transpose M_d^t. It is easy to see that each element a_{ij} of

$$A_d = M_d\, M_d^t = (a_{ij})$$

provides the number of common vertices between the d-face γ_i and the d-face γ_j. When this number is greater or equal than d, there is a common $(d-1)$-face shared between γ_i and γ_j.

In order to guarantee that all $(d-1)$-faces can be discovered by this method, a cellular decomposition of the whole $\mathbb{E}^d$ must be maintained, including both *solid* cells, i.e. the decomposition of the interior space, and *empty* cells, corresponding to a decomposition of the exterior space.

Exterior space of a block diagram

⟨ Exterior space of a block diagram 6 ⟩ ≡
```
""" Exterior space of a block diagram """
def exteriorCells(diagram):
   V,CV = diagram
   minVert, maxVert = min(V), max(V)
   d = len(V[0])
   outchain = [[] for k in range(2*d)]
   for k,v in enumerate(V):
      for h in range(d):
         if v[h] == minVert[h]: outchain[h] += [k]
         if v[h] == maxVert[h]: outchain[h+d] += [k]
   return outchain
◇
```
Macro referenced in 7a.

The aim of computing che chain of exterior cells is associated to the computation of of the $(d-1)$-skeleton, in turn needed for the computation of the boundary and coboundary operators. Look to Section 5.5 for a worked example.

3.2 Promoting local upgrades in all dimensions

4 Library export

4.1 Exporting the library

"larlib/larlib/sysml.py" 7a ≡
```
""" sysml library """
from larlib import *
DRAW = COMP([VIEW,STRUCT,MKPOLS])
```

⟨ To compute the boundary (d-1)-chain of a given d-chain 13b ⟩
⟨ Diagram initialization (non-uniform sizing) 2a ⟩
⟨ Boundary cells $(3D \to 2D)$ computation 2c ⟩
⟨ Interior partitions $(3D \to 2D)$ computation 3a ⟩

⟨ Diagram scaling to sized cuboid 2b ⟩
⟨ Drawing numbers of cells ? ⟩
⟨ Subdiagram to diagram mapping 3b ⟩
⟨ Exterior space of a block diagram 6 ⟩
⟨ Place vertices of two LAR models in a common space 14a ⟩
◇

5 Tests

5.1 Diagram initialization

"test/py/sysml/test01.py" 7b ≡

```
""" testing initial steps of Assembly Diagram construction """
from larlib import *

shape = [1,2,2]
sizePatterns = [[1],[2,1],[0.8,0.2]]
diagram = assemblyDiagramInit(shape)(sizePatterns)
print "\n diagram =",diagram
VIEW(SKEL_1(STRUCT(MKPOLS(diagram))))

VV,EV,FV,CV = gridSkeletons(shape)
boundaryFaces = lar2boundaryFaces(CV,FV)
interiorFaces = list(set(range(len(FV))).difference(boundaryFaces))
print "\n boundary faces =",boundaryFaces
print "\n interior faces =",interiorFaces
diagram1 = unitDiagram(diagram)
VIEW(SKEL_1(STRUCT(MKPOLS(diagram1))))

hpc = SKEL_1(STRUCT(MKPOLS(diagram1)))
V = diagram1[0]
hpc = cellNumbering((V,FV),hpc)(interiorFaces,YELLOW,.5)
VIEW(hpc)
hpc = cellNumbering((V,EV),hpc)([f for f in interiorFaces],GREEN,.4)
VIEW(hpc)
hpc = cellNumbering((V,VV),hpc)(range(len(VV)),RED,.3)
VIEW(hpc)
```

◇

5.2 Diagram merging

"test/py/sysml/test02.py" 8a ≡

```
""" definition and merging of two diagrams into a single diagram """
from larlib import *
```

```
master = assemblyDiagramInit([2,2,2])([[.4,.6],[.4,.6],[.4,.6]])
diagram = assemblyDiagramInit([3,3,3])([[.4,.2,.4],[.4,.2,.4],[.4,.2,.4]])
VIEW(SKEL_1(STRUCT([DRAW(master),T(2)(1),DRAW(diagram)])))

hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),WHITE,.5)
VIEW(hpc)

master = diagram2cell(diagram,master,7)
VIEW(SKEL_1(STRUCT( MKPOLS(master) )))

◇
```

5.3 Diagram visualization

"test/py/sysml/test03.py" 8b ≡

```
""" definition and merging of two diagrams into a single diagram """
from larlib import *

master = assemblyDiagramInit([2,2,2])([[.4,.6],[.4,.6],[.4,.6]])
diagram = assemblyDiagramInit([3,3,3])([[.4,.2,.4],[.4,.2,.4],[.4,.2,.4]])

VV,EV,FV,CV = gridSkeletons([2,2,2])
V,CV = master
hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(CV)),CYAN,.5)
VIEW(hpc)

master = diagram2cell(diagram,master,7)
VIEW(SKEL_1(STRUCT( MKPOLS(master) )))

VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(larFacets(master))))

masterBoundaryFaces = boundaryOfChain(CV,FV)([7])
diagramBoundaryFaces = lar2boundaryFaces(CV,FV)
◇
```

5.4 progressive refinement of a block diagram

In this example, a step-by step generation of a simple apartment is produced, using `assemblyDiagramInit` to produce a block diagram of given `shape` and `size`, the `cellNumbering` function to generate an *hpc* value with the numbers of 3-cells in the current "master" diagram, the `diagram2cell` function to map and merge a `diagram` into a `cell` of the `master`.

The construction process is visualised in Figure 3.

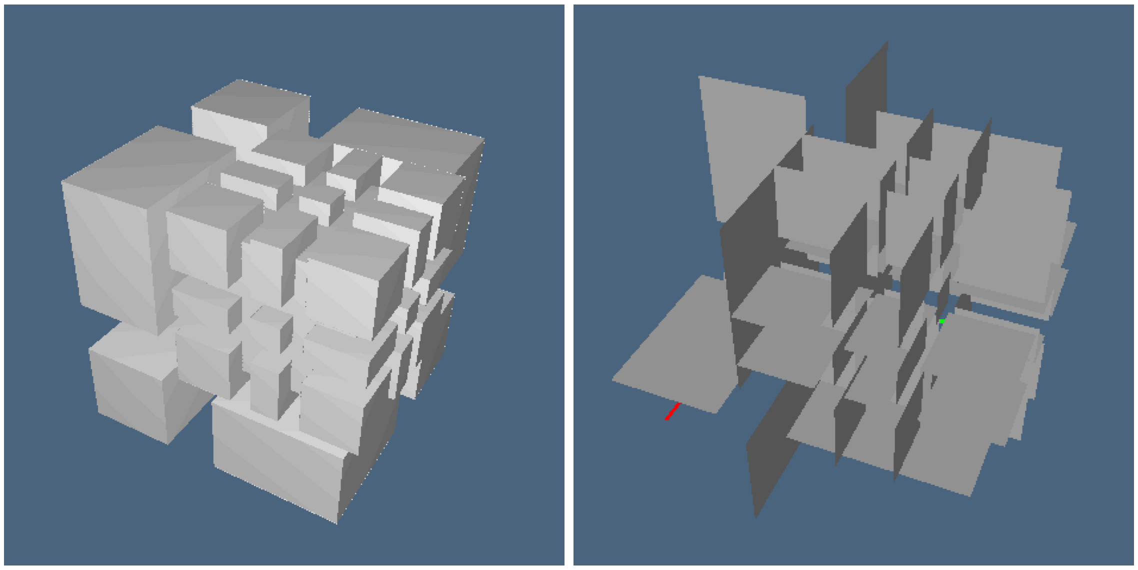

Figure 2: Example of a geometry diagram merged in a master diagram

Remember that in `lar-cc` the numbering of cells in a model is 0-based (like in python). Conversely, in `pyplasm` the numbering of cells (for example of vertex indices in `MKPOL`) is 1-based, like in Fortran or MATLAB.

"test/py/sysml/test04.py" 9 ≡
```
    """ progressive refinement of a block diagram """
    from larlib import *

    master = assemblyDiagramInit([5,5,2])([[.3,3.2,.1,5,.3],[.3,4,.1,2.9,.3],[.3,2.7]])
    V,CV = master
    hpc = SKEL_1(STRUCT(MKPOLS(master)))
    hpc = cellNumbering (master,hpc)(range(len(CV)),CYAN,2)
    VIEW(hpc)

    toRemove = [13,33,17,37]
    master = V,[cell for k,cell in enumerate(CV) if not (k in toRemove)]
    DRAW(master)

    hpc = SKEL_1(STRUCT(MKPOLS(master)))
    hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
    VIEW(hpc)

    toMerge = 29
    cell = MKPOL([master[0],[[v+1 for v in  master[1][toMerge]]],None])
    VIEW(STRUCT([hpc,cell]))
```

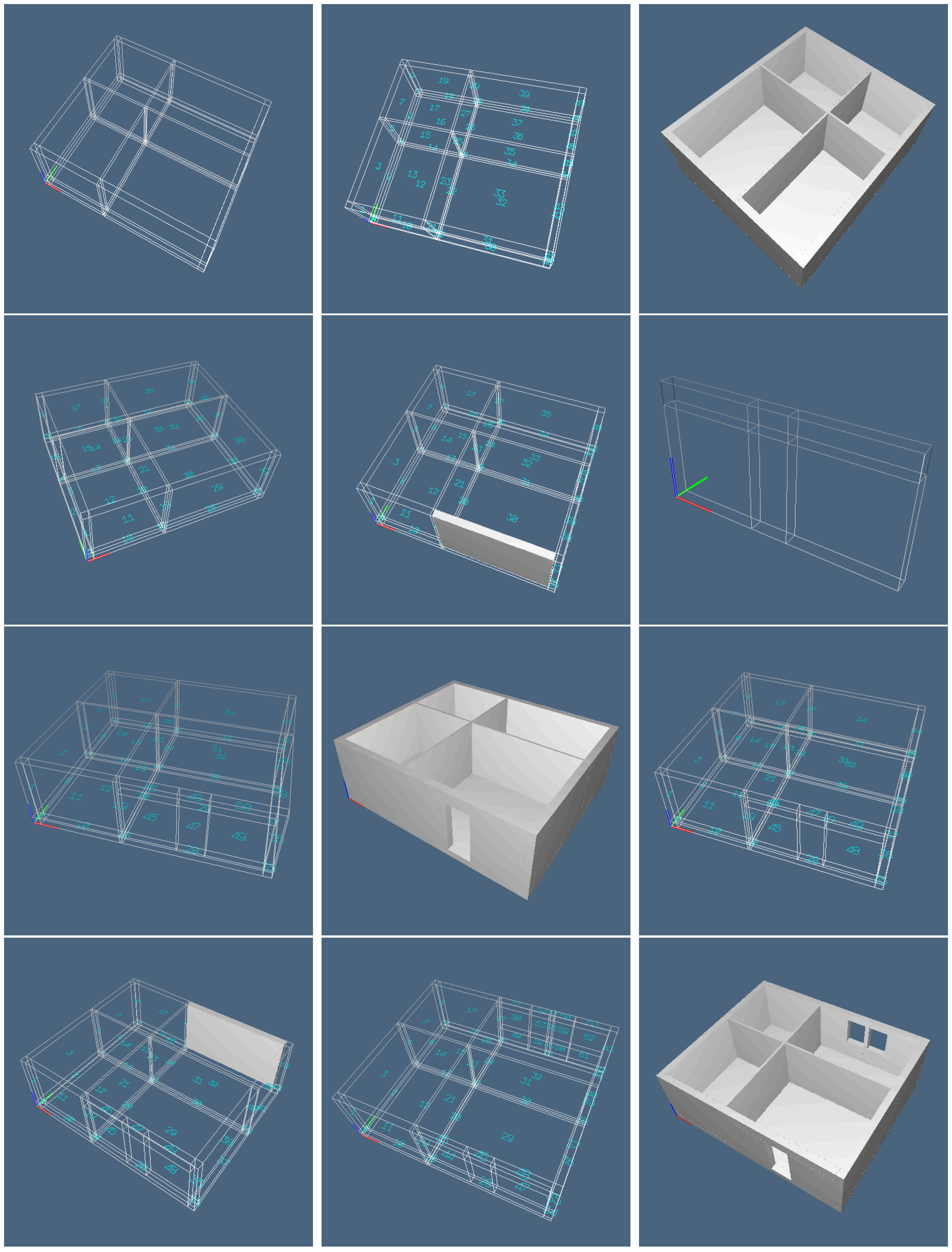

Figure 3: The construction process of the `master` block diagram built by the example `test/py/sysml/test4.py` of Section 5.4.

```
diagram = assemblyDiagramInit([3,1,2])([[2,1,2],[.3],[2.2,.5]])
master = diagram2cell(diagram,master,toMerge)
hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
VIEW(hpc)

toRemove = [47]
master = master[0], [cell for k,cell in enumerate(master[1]) if not (k in toRemove)]
DRAW(master)

hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
VIEW(hpc)

toMerge = 34
cell = MKPOL([master[0],[[v+1 for v in  master[1][toMerge]]],None])
VIEW(STRUCT([hpc,cell]))

diagram = assemblyDiagramInit([5,1,3])([[1.5,0.9,.2,.9,1.5],[.3],[1,1.4,.3]])
master = diagram2cell(diagram,master,toMerge)
hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
VIEW(hpc)

toRemove = [53,59]
master = master[0], [cell for k,cell in enumerate(master[1]) if not (k in toRemove)]
DRAW(master)
```
◇

5.5 Using the cochain of exterior cells

Here we develop the same example given above, but using also a cochain of empty cells, in order to be able to extract the boundary and coboundary operators of the cell decompositions. The `exteriorChain` of the `master` diagram is first computed after the `master` initialisation, and later updated with cells defined as empty

"test/py/sysml/test05.py" 12 ≡
```
""" boundary extraction of a block diagram """
from larlib import *

DRAW = COMP([VIEW,STRUCT,MKPOLS])

master = assemblyDiagramInit([5,5,2])([[.3,3.2,.1,5,.3],[.3,4,.1,2.9,.3],[.3,2.7]])
diagram1 = assemblyDiagramInit([3,1,2])([[2,1,2],[.3],[2.2,.5]])
diagram2 = assemblyDiagramInit([5,1,3])([[1.5,0.9,.2,.9,1.5],[.3],[1,1.4,.3]])
```

```
hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
VIEW(hpc)

master = diagram2cell(diagram2,master,39)
master = diagram2cell(diagram1,master,31)

hpc = SKEL_1(STRUCT(MKPOLS(master)))
hpc = cellNumbering (master,hpc)(range(len(master[1])),CYAN,2)
VIEW(hpc)

emptyChain = [17,13,32,36,52,58,65]
solidCV = [cell for k,cell in enumerate(master[1]) if not (k in emptyChain)]
DRAW((master[0],solidCV))

exteriorCV =  [cell for k,cell in enumerate(master[1]) if k in emptyChain]
exteriorCV += exteriorCells(master)
CV = solidCV + exteriorCV
V = master[0]
FV = [f for f in larFacets((V,CV),3,len(exteriorCV))[1] if len(f) >= 4]
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS((V,FV))))

BF = boundaryCells(solidCV,FV)
boundaryFaces = [FV[face] for face in BF]
B_Rep = V,boundaryFaces
VIEW(EXPLODE(1.1,1.1,1.1)(MKPOLS(B_Rep)))
VIEW(STRUCT(MKPOLS(B_Rep)))

⟨Transform the LAR boundary model in a triangle model 13a⟩
◇
```

Transform the LAR boundary model in a triangles model The transformation from a boundary representation made by general 2D convex faces to a set of triangle faces is provided below.

⟨Transform the LAR boundary model in a triangle model 13a⟩ ≡

```
verts, triangles = quads2tria(B_Rep)
B_Rep = V,boundaryFaces
VIEW(EXPLODE(1.1,1.1,1.1)(MKPOLS((verts, triangles))))
VIEW(STRUCT(MKPOLS((verts, triangles))))
◇
```

Macro referenced in 12.

A Utilities

⟨ To compute the boundary (d-1)-chain of a given d-chain 13b ⟩ ≡

```
def boundaryOfChain(cells,facets):
   csrBoundaryMat = larBoundary(cells,facets)
   csrChain = zeros((len(cells),1))
   def boundaryOfChain0(chain):
      for cell in chain:  csrChain[cell,0]=1.0
      csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain)
      boundaryCells = [k for k,val in enumerate(csrBoundaryChain.tolist())
                     if val == [1.0]]
      return boundaryCells
   return boundaryOfChain0
◇
```

Macro referenced in 7a.

A.1 Initial import of modules

Initial import of modules

⟨ Initial import of modules 13c ⟩ ≡

```
""" Initial import of modules """
from lar2psm import *
◇
```

Macro never referenced.

A.2 Reordering of vertex coordinates

A global reordering of vertex coordinates is executed as the first step of the Boolean algorithm, in order to eliminate the duplicate vertices, by substituting duplicate vertex copies (coming from two close points) with a single instance.

Two dictionaries are created, then merged in a single dictionary, and finally split into three subsets of (vertex,index) pairs, with the aim of rebuilding the input representations, by making use of a novel and more useful vertex indexing.

The union set of vertices is finally reordered using the three subsets of vertices belonging (a) only to the first argument, (b) only to the second argument and (c) to both, respectively denoted as V_1, V_2, V_{12}. A top-down description of this initial computational step is provided by the set of macros discussed in this section.

⟨ Place vertices of two LAR models in a common space 14a ⟩ ≡

```
""" Place vertices of two LAR models in a common space """
⟨ Initial indexing of vertex positions 14b ⟩
⟨ Merge two dictionaries with keys the point locations 15a ⟩
```

⟨ Filter the common dictionary into three subsets 15b ⟩
⟨ Compute an inverted index to reorder the vertices of arguments 16a ⟩
⟨ Return the single reordered pointset and the two *d*-cell arrays 16b ⟩
◇

Macro referenced in 7a.

A.2.1 Re-indexing of vertices

Initial indexing of vertex positions The input LAR models are located in a common space by (implicitly) joining `V1` and `V2` in a same array, and (explicitly) shifting the vertex indices in `CV2` by the length of `V1`.

⟨ Initial indexing of vertex positions 14b ⟩ ≡

```
from collections import defaultdict, OrderedDict

def vertexSieve(model1, model2):
   V1,CV1 = model1; V2,CV2 = model2
   n = len(V1); m = len(V2)
   def shift(CV, n):
      return [[v+n for v in cell]for cell in CV]
   CV2 = shift(CV2,n)
```
◇

Macro referenced in 14a.

Merge two dictionaries with point location as keys Since currently `CV1` and `CV2` point to a set of vertices larger than their initial sets `V1` and `V2`, we index the set `V1` ∪ `V2` using a Python `defaultdict` dictionary, in order to avoid errors of "missing key". As dictionary keys, we use the string representation of the vertex position vector provided by the `vcode` function given in the Appendix.

⟨ Merge two dictionaries with keys the point locations 15a ⟩ ≡

```

   vdict1 = defaultdict(list)
   for k,v in enumerate(V1): vdict1[vcode(v)].append(k)
   vdict2 = defaultdict(list)
   for k,v in enumerate(V2): vdict2[vcode(v)].append(k+n)

   vertdict = defaultdict(list)
   for point in vdict1.keys(): vertdict[point] += vdict1[point]
   for point in vdict2.keys(): vertdict[point] += vdict2[point]
```
◇

Macro referenced in 14a.

Example of string coding of a vertex position The position vector of a point of real coordinates is provided by the function `vcode`. An example of coding is given below. The *precision* of the string representation can be tuned at will.

```
>>> vcode([-0.011660381062724849, 0.297350056848685860])
'[-0.0116604, 0.2973501]'
```

Filter the common dictionary into three subsets `Vertdict`, dictionary of vertices, uses as key stye position vectors of vertices coded as string, and as values the list of integer indices of vertices on the given position. If the point position belongs either to the first or to second argument only, it is stored in `case1` or `case2` lists respectively. If the position (`item.key`) is shared between two vertices, it is stored in `case12`. The variables `n1`, `n2`, and `n12` remember the number of vertices respectively stored in each repository.

⟨ Filter the common dictionary into three subsets 15b ⟩ ≡

```
        case1, case12, case2 = [],[],[]
        for item in vertdict.items():
           key,val = item
           if len(val)==2:  case12 += [item]
           elif val[0] < n: case1 += [item]
           else: case2 += [item]
        n1 = len(case1); n2 = len(case12); n3 = len(case2)
```
◇

Macro referenced in 14a.

Compute an inverted index to reorder the vertices of Boolean arguments The new indices of vertices are computed according with their position within the storage repositories `case1`, `case2`, and `case12`. Notice that every `item[1]` stored in `case1` or `case2` is a list with only one integer member. Two such values are conversely stored in each `item[1]` within `case12`.

⟨ Compute an inverted index to reorder the vertices of arguments 16a ⟩ ≡

```
        invertedindex = list(0 for k in range(n+m))
        for k,item in enumerate(case1):
           invertedindex[item[1][0]] = k
        for k,item in enumerate(case12):
           invertedindex[item[1][0]] = k+n1
           invertedindex[item[1][1]] = k+n1
        for k,item in enumerate(case2):
           invertedindex[item[1][0]] = k+n1+n2
```
◇

Macro referenced in 14a.

A.2.2 Re-indexing of d-cells

Return the single reordered pointset and the two d-cell arrays We are now finally ready to return two reordered LAR models defined over the same set `V` of vertices, and where (a) the vertex array `V` can be written as the union of three disjoint sets of points C_1, C_{12}, C_2; (b) the d-cell array `CV1` is indexed over $C_1 \cup C_{12}$; (b) the d-cell array `CV2` is indexed over $C_{12} \cup C_2$.

The `vertexSieve` function will return the new reordered vertex set $V = (V_1 \cup V_2) \setminus (V_1 \cap V_2)$, the two renumbered s-cell sets `CV1` and `CV2`, and the size `len(case12)` of $V_1 \cap V_2$.

⟨Return the single reordered pointset and the two d-cell arrays 16b⟩ ≡

```
        V = [eval(p[0]) for p in case1] + [eval(p[0]) for p in case12] + [eval(
                 p[0]) for p in case2]
        CV1 = [sorted([invertedindex[v] for v in cell]) for cell in CV1]
        CV2 = [sorted([invertedindex[v] for v in cell]) for cell in CV2]
        return V, CV1, CV2, len(case12)
```
◇

Macro referenced in 14a.

References

[CL13] CVD-Lab, *Linear algebraic representation*, Tech. Report 13-00, Roma Tre University, October 2013.