Modeling Geometry with Assemblies in SysML *

May 19, 2014

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]. May 19, 2014

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 7b.

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 7b.

2.2 Cell numbering

Drawing numbers of cells

⟨ Drawing numbers of cells 2c ⟩ ≡
```
""" Drawing numbers of cells """
def cellNumbering (larModel,hpcModel):
   V,CV = larModel
   def cellNumbering0 (cellSubset,color=WHITE,scalingFactor=1):
      text = TEXTWITHATTRIBUTES (TEXTALIGNMENT='centre', TEXTANGLE=0,
                     TEXTWIDTH=0.1*scalingFactor,
                     TEXTHEIGHT=0.2*scalingFactor,
                     TEXTSPACING=0.025*scalingFactor)
      hpcList = [hpcModel]
      for cell in cellSubset:
         point = CCOMB([V[v] for v in CV[cell]])
         hpcList.append(T([1,2,3])(point)(COLOR(color)(text(str(cell)))))
      return STRUCT(hpcList)
   return cellNumbering0
```
◇

Macro referenced in 7b.

2.3 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 3a ⟩ ≡

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

Macro referenced in 7b.

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 3b ⟩ ≡

```
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 7b.

2.4 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 4 ⟩ ≡

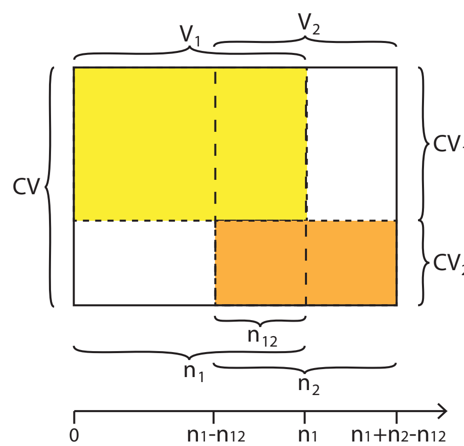

Figure 1: Structure of the characteristic matrix $M(CV)$ afre the merge of two LAR models, and identification of the common vertices.

⟨ 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)
   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 7b.

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 4.

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 7a ⟩ ≡

```
""" 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 7b.

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

"lib/py/sysml.py" 7b ≡

⟨ Initial import of modules 13b ⟩
⟨ To compute the boundary (d-1)-chain of a given d-chain 13a ⟩
⟨ Diagram initialization (non-uniform sizing) 2a ⟩
⟨ Boundary cells ($3D \to 2D$) computation 3a ⟩
⟨ Interior partitions ($3D \to 2D$) computation 3b ⟩
⟨ Diagram scaling to sized cuboid 2b ⟩
`from myfont import *`
⟨ Drawing numbers of cells 2c ⟩
⟨ Subdiagram to diagram mapping 4 ⟩
⟨ Exterior space of a block diagram 7a ⟩
◇

5 Tests

5.1 Diagram initialization

"test/py/sysml/test01.py" 7c ≡

```
""" testing initial steps of Assembly Diagram construction """
⟨ Initial import of modules 13b ⟩
from sysml 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)([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" 8 ≡

```
""" definition and merging of two diagrams into a single diagram """
⟨Initial import of modules 13b⟩
from sysml 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" 9a ≡

```
""" definition and merging of two diagrams into a single diagram """
⟨Initial import of modules 13b⟩
from sysml 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)
◇
```

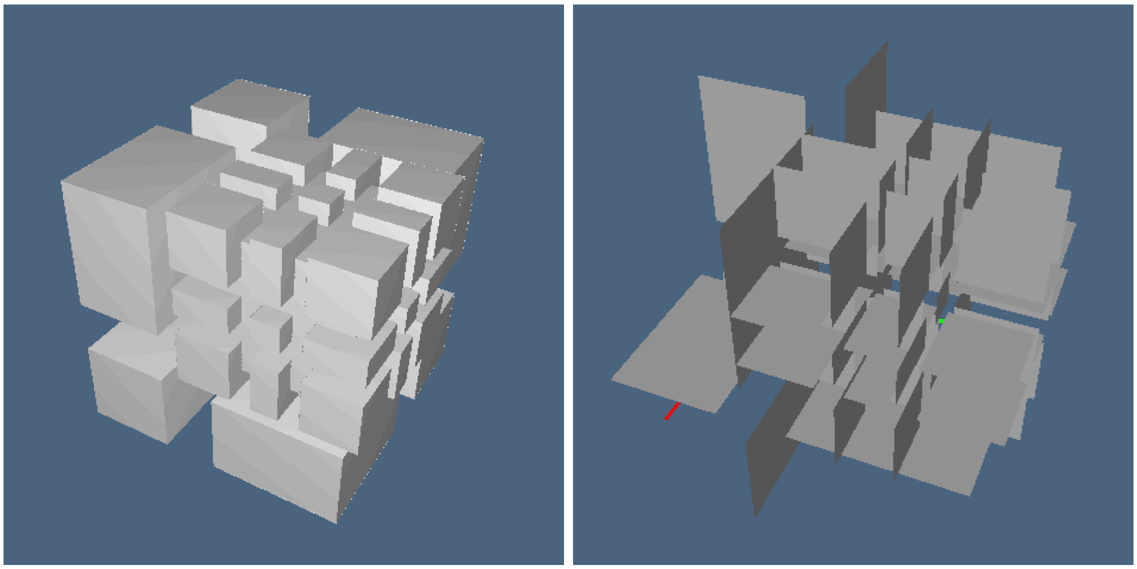

Figure 2: Example of a geometry diagram merged in a master diagram

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.

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" 9b ≡

```
""" progressive refinement of a block diagram """
⟨Initial import of modules 13b⟩
from sysml import *
DRAW = COMP([VIEW,STRUCT,MKPOLS])
```

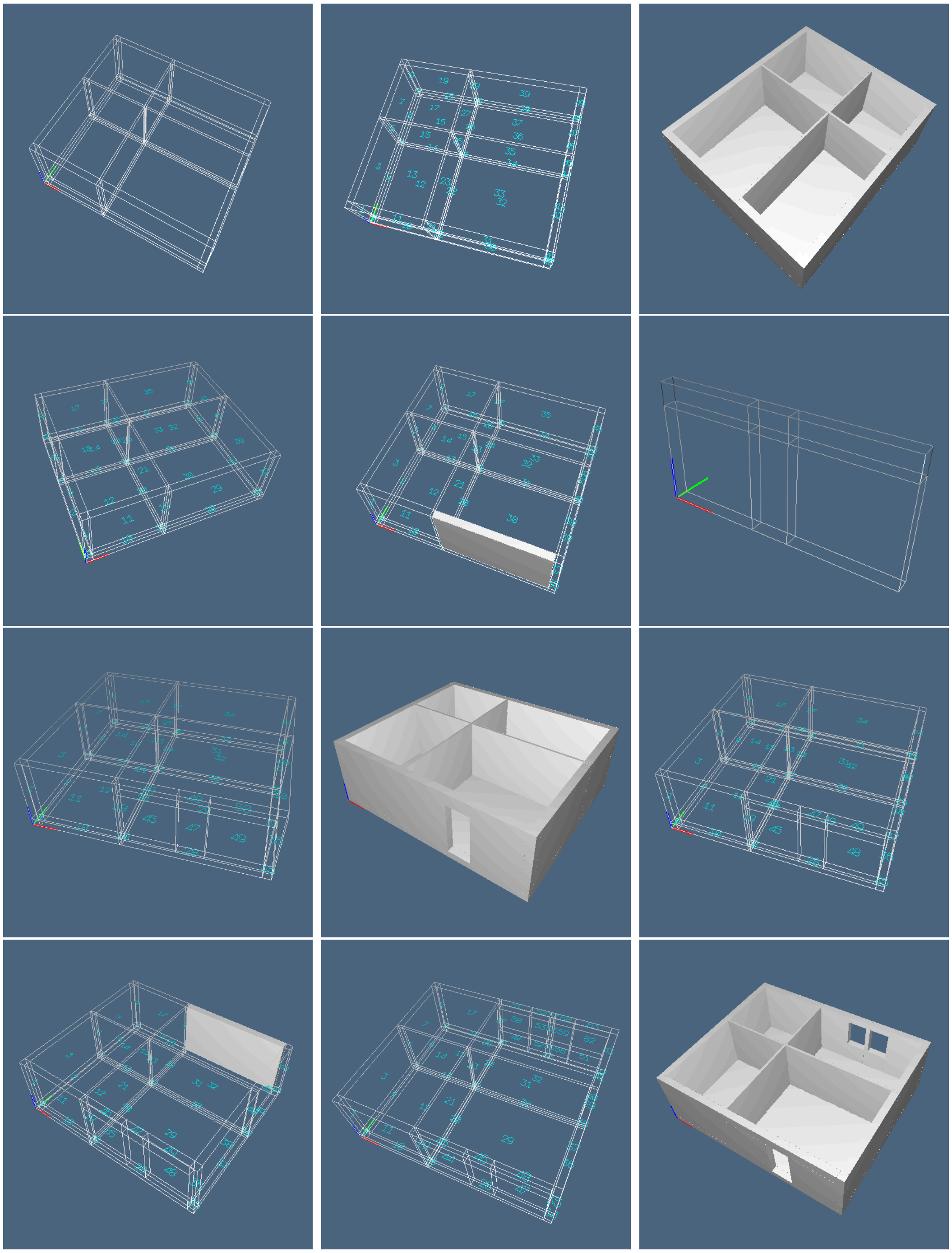

Figure 3: The construction process of the `master` block diagram built by the example `test/py/sysml/test4.py` of Section 5.4.

```
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]))

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 """
⟨Initial import of modules 13b⟩
from sysml 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(1.5,1.5,1.5)(MKPOLS(B_Rep)))
VIEW(STRUCT(MKPOLS(B_Rep)))
◇
```

A Utilities

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

```
def boundaryOfChain(cells,facets):
   csrBoundaryMat = boundary(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 7b.

A.1 Initial import of modules

Initial import of modules

⟨ Initial import of modules 13b ⟩ ≡

```
from pyplasm import *
from scipy import *
import os,sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from lar2psm import *
from simplexn import *
from larcc import *
from largrid import *
from mapper import *
from boolean import *
◇
```

Macro referenced in 7bc, 8, 9ab, 12.

References

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