

Modeling Geometry with Assemblies in SysML *

May 17, 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 is used to specify an operable description of the 3D geometry of a system part.

Contents

1	Introduction	1
1.1	bbbbbbbbb	1
2	Implementation	1
2.1	Diagram initialization	1
2.2	Cell numbering	2
2.3	Diagram segmentation	2
2.4	Subdiagram mapping	2
3	Library export	3
3.1	Exporting the library	3
4	Tests	3
4.1	Diagram initialization	3
4.2	Diagram merging	4
4.3	Diagram visualization	4
A	Utilities	4
A.1	Initial import of modules	5

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [?].
May 17, 2014

1 Introduction

1.1 bbbbbb

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.

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

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".

```
@D Diagram initialization (non-uniform sizing) @""" Diagram initialization """ def
assemblyDiagramInit (shape): def assemblyDiagram (quoteList): print "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)returnverts,CVreturnassemblyDiagram@
```

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

```
@D Diagram scaling to sized cuboid @""" Diagram scaling to given size """ def unit-
Diagram(diagram, size=[1,1,1]): V,CV = diagram print "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,CVreturndiagram@
```

2.2 Cell numbering

Drawing numbers of cells @D Drawing numbers of cells @""" Drawing numbers of cells

```
""" def cellNumbering (larModel,hpcModel): V,CV = larModel def cellNumbering0 (cell-
Subset,color=WHITE,scalingFactor=1): text = TEXTWITHATTRIBUTES (TEXTAL-
IGNMENT='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 @
```

2.3 Diagram segmentation

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

```
@D Boundary cells ( $3D \rightarrow 2D$ ) computation @def lar2boundaryFaces(CV,FV): """
Boundary cells computation """ return boundaryCells(CV,FV) @
```

Interior partitions ($3D \rightarrow 2D$) computation The indices of the boundary 2-cells are returned in `boundarychain2D`, and subtracted from the set $\{0, 1, \dots, |E| - 1\}$ in order to return the indices of the `interiorCells`. @D Interior partitions ($3D \rightarrow 2D$) computation @def lar2InteriorFaces(CV,FV): """ Boundary cells computation """ boundarychain2D = boundaryCells(CV,FV) totalChain2D = range(len(FV)) interiorCells = set(totalChain2D).difference(boundarychain2D) return interiorCells @

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 ??.

```
@D Subdiagram to diagram mapping @ @j 3D window to viewport transformation @i
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)
```

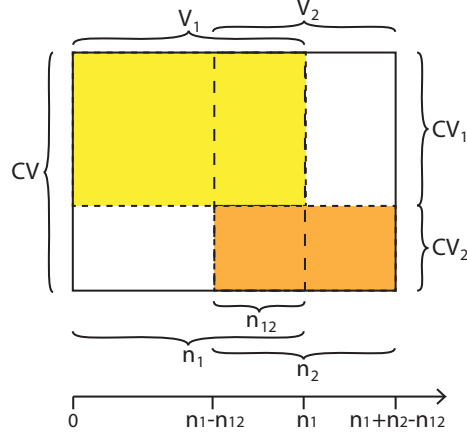


Figure 1: Structure of the characteristic matrix $M(CV)$ after the merge of two LAR models, and identification of the common vertices.

```
def checkInclusion(V,theCell,newRange): print "in" print "theCell =",theCell print
"newRange =",newRange theVerts = [V[v] for v in theCell] theMin, theMax = min(theVerts),
max(theVerts) theCell += [v for v in newRange if (theMin != V[v]) and (V[v] != theMax)]
print "theCell =",theCell print "out" return theCell
```

```
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 @
```

```
@D Possible append of new vertices to a 3-cell @
```

```
@
```

3D window to viewport transformation @D 3D window to viewport transformation

```
@""" 3D window to viewport transformation """ def diagram2cellMatrix(diagram): def di-
agramToCellMatrix0(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
"window3D =",wdw print "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 "mat =",mat return mat return diagramToCellMatrix0 @
```

3 Library export

3.1 Exporting the library

@O lib/py/sysml.py @i Initial import of modules @i @i To compute the boundary (d-1)-chain of a given d-chain @i @i Diagram initialization (non-uniform sizing) @i @i Boundary cells ($3D \rightarrow 2D$) computation @i @i Interior partitions ($3D \rightarrow 2D$) computation @i @i Diagram scaling to sized cuboid @i from myfont import * @i Drawing numbers of cells @i @i Subdiagram to diagram mapping @i @

4 Tests

4.1 Diagram initialization

```
@O test/py/sysml/test01.py @""" testing initial steps of Assembly Diagram construction
""" @i Initial import of modules @i from sysml import *
    shape = [1,2,2] sizePatterns = [[1],[2,1],[0.8,0.2]] diagram = assemblyDiagramInit(shape)(sizePatterns)
print "diagram =",diagram VIEW(SKEL1(STRUCT(MKPOLS(diagram))))
    VV,EV,FV,CV = gridSkeletons(shape) boundaryFaces = lar2boundaryFaces(CV,FV)
interiorFaces = list(set(range(len(FV)).difference(boundaryFaces)) print "boundary faces
=",boundaryFaces print "interior faces =",interiorFaces diagram1 = unitDiagram(diagram)
VIEW(SKEL1(STRUCT(MKPOLS(diagram1))))
    hpc = SKEL1(STRUCT(MKPOLS(diagram1)))V = diagram1[0]hpc = cellNumbering((V,FV),hpc)(in
cellNumbering((V,EV),hpc)([for fininteriorFaces], GREEN, .4)VIEW(hpc)hpc = cellNumbering((V,VV
@
```

4.2 Diagram merging

```
@O test/py/sysml/test02.py @""" definition and merging of two diagrams into a single
diagram """ @i Initial import of modules @i from sysml import *
    master = assemblyDiagramInit([2,2,2])([[.4,.6],[.4,.6],[.4,.6]]) diagram = assemblyDia-
gramInit([3,3,3])([[.4,.2,.4],[.4,.2,.4],[.4,.2,.4]]) VIEW(SKEL1(STRUCT([DRAW(master),T(2)(1),DRAW(di
hpc = SKEL1(STRUCT(MKPOLS(master)))hpc = cellNumbering(master,hpc)(range(len(master[1]))
    master = diagram2cell(diagram,master,7) VIEW(SKEL1(STRUCT(MKPOLS(master))))
@
```

4.3 Diagram visualization

```
@O test/py/sysml/test03.py @""" definition and merging of two diagrams into a single
diagram """ @i Initial import of modules @i from sysml import *
    master = assemblyDiagramInit([2,2,2])([[.4,.6],[.4,.6],[.4,.6]]) diagram = assemblyDia-
gramInit([3,3,3])([[.4,.2,.4],[.4,.2,.4],[.4,.2,.4]])
```

```

VV,EV,FV,CV = gridSkeletons([2,2,2]) V,CV = master hpc = SKEL1(STRUCT(MKPOLS(master)))hpc
cellNumbering(master,hpc)(range(len(CV)),CYAN,.5)VIEW(hpc)
master = diagram2cell(diagram,master,7) VIEW(SKEL1(STRUCT(MKPOLS(master))))
VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLs(larFacets(master))))
masterBoundaryFaces = boundaryOfChain(CV,FV)([7]) diagramBoundaryFaces = lar2boundaryFaces(CV)
@

```

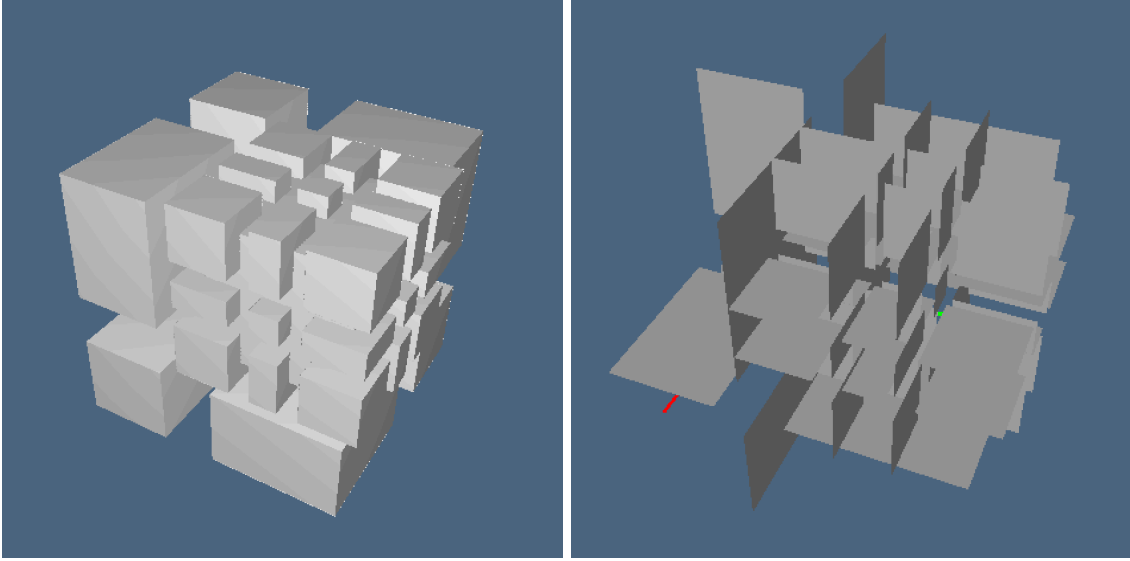


Figure 2: Example of a geometry diagram merged in a master diagram

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`.

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.

```

@O test/py/sysml/test04.py @""" progressive refinement of a block diagram """ @i Initial
import of modules @i 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]]) V,CV =
master hpc = SKEL1(STRUCT(MKPOLS(master)))hpc = cellNumbering(master,hpc)(range(len(CV)),C
toRemove = [13,33,17,37] master = V,[cell for k,cell in enumerate(CV) if not (k in
toRemove)] DRAW(master)
hpc = SKEL1(STRUCT(MKPOLS(master)))hpc = cellNumbering(master,hpc)(range(len(master[1]))

```

```

    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 = SKEL1(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 = SKEL1(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 = SKEL1(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) @

```

A Utilities

```

@D To compute the boundary (d-1)-chain of a given d-chain @ 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 @

```

A.1 Initial import of modules

```

Initial import of modules @D Initial import of modules @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 * @

```