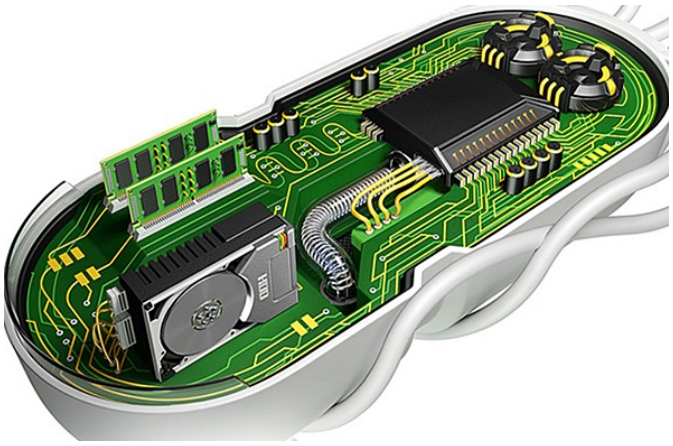
Cello 2.0

A Design Automation Tool for Genetic Circuits

Technical Report and Documentation



Revised: May 2018

This work was supported by the Synergistic Discovery and Design (SD2) grant from the Defense Advanced Research Projects Agency (DARPA).

Table of Contents

Contributors:

- Christopher A. Voigt, *Massachusetts Institute of Technology (MIT)* – (cavoigt@gmail.com)
  - Advisor (Chief Architect)

- Vincent Mirian, *Massachusetts Institute of Technology (MIT)* – (vince.mirian@gmail.com)
  - Yosys
  - maxFanout
  - CL_RC
  - HMetis (Integration)
  - GPCC_BASE
  - GPCC_SCIP_BASE
  - GPCC_SUGARM_BASE
  - GPCC_GRID
  - Cello_JY_TP (Enhanced for multi-cell)
- Jai Padmakumar, *Massachusetts Institute of Technology (MIT)* – (jaip217@gmail.com)
  - HMetis (Research)
- Shuyi Zhang, *Massachusetts Institute of Technology (MIT)* – (shuyi@mit.edu)
  - Cello_JY_TP (Preliminary Developments)
- Jonghyeon Shin, *Massachusetts Institute of Technology (MIT)* – (shinx097@mit.edu)
  - Cello_JY_TP (Preliminary Developments)

1 Introduction

Cello 2.0 is a design automation tool for genetic circuits. Cello 2.0 is built using Porus (**TODO REF**). This document describes the application(s), stage(s) and algorithm(s) present in the tool.

2 Libraries

The Poros framework is equipped with libraries. The Cello project is enhanced with libraries written in the JAVA programming language and are found in the *external_jars* folder of the *Cello* project. Update the User Library as described in Chapter 2 of the Poros documentation.

Filename	**Description**
guava-23.0.jar[1]	Guava is a set of core libraries that includes new collection types (such as multimap and multiset), immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection, string processing, and much more!

Table 1: List of libraries in the Cello project

Table 1 shows a list of libraries in the Cello project. The first column of the table contains the filename and the second column contains a description of the library.

Cello.userlibraries is a library configuration file in the root directory of the *Cello* project. The configuration file can be imported to setup the Eclipse project. To import the file:

1) Windows > Preferences > Java > Build Path > User Libraries

2) Import…

3) Select *Cello.userlibraries*

Cello.userlibraries can also be updated for future use by exporting the user library configuration.

1 https://github.com/google/guava/wiki/Release23

3 Applications

This Chapter presents the applications in the Cello project.

3.1 DNACompiler

The *DNACompiler* application outputs a DNA sequence using: 1) the user design, 2) the user design constraints, and, 3) the target data. This section describes: 1) the execution flow of the application, 2) the stage(s) used within the application, and, 3) the usage of the applications and its features.

The *applications/DNACompiler* folder contains the data for building the *DNACompiler* application using the Poros framework, such as the *INFILE* for adding the stages to the application.

3.1.1 Execution

The command line arguments of this application are identical to the Poros command line arguments described in Chapter 10 of the Poros documentation.

The following is the execution flow of the *DNACompiler* application:

1) the *logicSynthesis* stage (Section 4.1)
2) the *logicOptimization* stage (Section 4.2)
3) the *clustering* stage (Section 4.3)
4) the *partitioning* stage (Section 4.4)
5) the *placing* stage (Section 4.5)
6) the *technologyMapping* stage (Section 4.6)

Stage	Default Algorithm
logicSynthesis	*Yosys* (Section 4.1.1)
logicOptimization	*maxFanout* (Section 4.2.1)
clustering	*CL_RC* (Section 4.3.1)
partitioning	*GPCC_SUGARM_BASE* (Section 4.4.3)
placing	*GPCC_GRID* (Section 4.5.1)
technologyMapping	*Cello_JY_TP* (Section 4.6.1)

Table 2: Summary of the default algorithms for each stage

Table 2 shows a summary of the default algorithms for each stage. The first column corresponds to the stage and the second column corresponds to the default algorithm. Each entry of the table shows the one-to-one mapping of the stage with its default algorithm.

3.1.2 sample-input

The following is a description of the sample-inputs for the *DNACompiler* application.

adder

The *adder* sample-input is a N-bit full adder[2]. The N-bit full adder is a primitive digital logic circuit that: 1) adds two N-bit values and a 1-bit carry-in value, and, 2) outputs the sums of the two N-bit values with a 1-bit carry-out value. The circuit is described in the *adder.v* file. The *WIDTH* parameter defines the number of bits (N) for the adder. The version of the target data is *Eco1C1G1T1*. The contents for this sample-input are in the *sample-input/DNACompiler/adder* folder of the *Cello* project.

ALU

The *ALU* sample-input is a N-bit Arithmetic Logic Unit (ALU[3]). An ALU is a core component found in a Central Processing Unit (CPU). The ALU is composed of several mathematical operations that are listed in the Verilog file (*alu.v*). The circuit is described in the *alu.v* file. The *WIDTH* parameter defines the number of bits (N) for the ALU. The version of the target data is *Eco1C1G1T1*. The contents for this sample-input are in the *sample-input/DNACompiler/ALU* folder of the *Cello* project.

*alu**X*** directories contain the ALU circuit targeting a spatial device with ***X*** cells.

HexDisplay

The *hexDisplay* sample-input is the decoder logic for a 7 Segment Display[4] (*hexDisplay.v*). This circuit takes a 4-bit input from 0x0 to 0xF and outputs the digital value for each segments of the display. The circuit is implemented with a case statement. The version of the target data is *Eco1C1G1T1*. The contents for this sample-input are in the *sample-input/DNACompiler/hexDisplay* folder of the *Cello* project.

*hexDiplay**X*** directories contain the hexDisplay circuit targeting a spatial device with ***X*** cells.

MD5

The *md5Core* sample-input is an N-bit MD5[5] hash function. The MD5 hash function is a cryptographic hash function used as a checksum to verify data integrity. The circuit is described in the *md5Core.v* file. The *WIDTH* parameter defines the number of bits (N) for the MD5 hash function. The version of the target data is *Eco1C1G1T1*. The contents for this sample-input are in the *sample-input/DNACompiler/md5Core* folder of the *Cello* project.

*MD5Core**X*** directories contain the MD5 circuit targeting a spatial device with ***X*** cells.

Systolic

The *systolic* sample-input is a 2D systolic array[6]. A systolic array is a homogeneous network of tightly coupled data processing units (DPUs). The dimensions are defined using the *ROW* and *COLUMN* parameters. For *systolic*,* the DPUs are implemented as a NOR gate. For *systolic*Diff*, the DPUs in: 1) the lower half (relative to the diagonal) of the systolic array are implemented as a NOR gate, 2) across

2 https://en.wikipedia.org/wiki/Adder_(electronics)#Full_adder
3 https://en.wikipedia.org/wiki/Arithmetic_logic_unit
4 https://en.wikipedia.org/wiki/Seven-segment_display
5 https://en.wikipedia.org/wiki/MD5
6 https://en.wikipedia.org/wiki/Systolic_array

the diagonal of the systolic array are implemented as an AND gate, and, 3) in the upper half (relative to the diagonal) of the systolic array are implemented as an XOR gate. The version of the target data is *Eco1C1G1T1*. The contents for this sample-input are in the *sample-input/DNACompiler/systolic* folder of the *Cello* project.

*systolic**ROW**b**COLUMN*** directories contain the systolic circuit with rows determined by **ROW** and column determined by **COLUMN**.

4 Stages

This Chapter presents the stages in the Cello project.

4.1 logicSynthesis

The purpose of the *logicSynthesis* stage is to translate the user design to a *Netlist*, and, assign a *nodeType* to the instances of the *ResultNetlistNodeData* of a *NetlistNode* class of the *Netlist*.

The *ResultNetlistNodeData* class is modified with a *nodeType* attribute and the methods for accessing this attribute. The *setNodeType(nodeType)* method sets the value of the *nodeType* attribute, and, the *getNodeType()* method gets the value of the *nodeType* attribute.

A list of valid *nodeType* values are defined in the *LSResults* class of the *results.logicSynthesis* package. Table 3 shows a summary of the reference types, values and descriptions for a *nodeType*. In the table, each row defines a valid value for the *nodeType* attribute. The first column defines the member reference in the *LSResults* class, the second column defines the values, and, the third column provides a description.

Reference Type	Value	Description
S_PRIMARYINPUT	PRIMARY_INPUT	A primary input node
S_PRIMARYOUTPUT	*PRIMARY_OUTPUT*	A primary output node
S_INPUT	INPUT	A input node
S_OUTPUT	*OUTPUT*	A output node
S_NOT	NOT	A node representing the NOT gate, its functional equivalence is *NOT(A)*
S_AND	AND	A node representing the AND gate, its functional equivalence is *A AND B*
S_NAND	NAND	A node representing the NAND gate, its functional equivalence is *NOT(A AND B)*
S_OR	OR	A node representing the OR gate, its functional equivalence is *A OR B*
S_NOR	NOR	A node representing the NOR gate, its functional equivalence is *NOT(A OR B)*
S_XOR	XOR	A node representing the XOR gate, its functional equivalence is *A XOR B*
S_XNOR	XNOR	A node representing the XNOR gate, its functional equivalence is *NOT(A XOR B)*
S_ANDNOT	ANDNOT	A node representing the ANDNOT gate, its functional equivalence is *A AND NOT B*
S_ORNOT	ORNOT	A node representing the ORNOT gate, its functional equivalence is *A OR NOT B*
S_MUX	MUX	A node representing the MUX component, its functional equivalence is (*A AND NOT(S)) OR (B AND S)*
S_AOI3	AOI3	A node representing the AOI3 gate, its functional equivalence is *NOT((A AND B) OR C)*
S_OAI3	OAI3	A node representing the OAI3 gate, its functional equivalence is *NOT((A OR B) AND C)*
S_AOI4	AOI4	A node representing the AOI4 gate, its functional equivalence is *NOT((A AND B) OR (C AND D))*
S_OAI4	OAI4	A node representing the OAI4 gate, its functional equivalence is *NOT((A OR B) AND*

		(C OR D))

Table 3 Summary of the reference types, values and descriptions for a nodeType.

4.1.1 Yosys

The Yosys algorithm uses a third party tool entitled Yosys Open SYnthesis Suite[7] (YOSYS). Additional details on Yosys Open SYnthesis Suite can be found in the *./Cello/Documentation/DNACompiler/logicSynthesis/Yosys* folder. The remainder of this section describes the Yosys algorithm.
The algorithm creates a *Netlist* instance using a user design written in the 2005 specification of the Verilog hardware description language (HDL)[8]. The algorithm uses the ABC[9] synthesis tool to assign the *nodeType* of the instances of the *ResultNetlistNodeData* of a *NetlistNode* class of the *Netlist*.

The algorithm has a single parameter named *Gates* with type *String*. This parameter signifies the set of gates to assign the *nodeType* attribute. The default value for this parameter is *NOR*. Note that, by default, the *NOT* gate is always present in the set of gates. Hence, a *Gates* parameter defined as *NOR* signifies to translate the user design to *NOR* and *NOT* gates.

Table 4 shows a summary of the values and descriptions for a valid *Gates* parameter. To list more than one value for the *Gates* parameter, separate the values using a comma (,). For example, a *Gates* parameter defined as *XNOR,OR* signifies to translate the user design to *NOT*, *XNOR* and *OR* gates.

Value	Description
NOT	A node representing the NOT gate, its functional equivalence is *NOT(A)*
AND	A node representing the AND gate, its functional equivalence is *A AND B*
NAND	A node representing the NAND gate, its functional equivalence is *NOT(A AND B)*
OR	A node representing the OR gate, its functional equivalence is *A OR B*
NOR	A node representing the NOR gate, its functional equivalence is *NOT*(*A OR B)*
XOR	A node representing the XOR gate, its functional equivalence is *A XOR B*
XNOR	A node representing the XNOR gate, its functional equivalence is *NOT(A XOR B)*
ANDNOT	A node representing the ANDNOT gate, its functional equivalence is *A AND NOT B*
ORNOT	A node representing the ORNOT gate, its functional equivalence is *A OR NOT B*

Table 4: Summary of the values and descriptions for a valid Gates parameter

7 http://www.clifford.at/yosys/
8 https://standards.ieee.org/findstds/standard/1364-2005.html
9 http://people.eecs.berkeley.edu/~alanmi/abc/

4.2 logicOptimization

Logic optimization is the process of finding an equivalent representation of the specified logic circuit under one or more specified constraints.

4.2.1 MaxFanout

The maxFanout algorithm limits the fanout of a *NetlistNode* in the *Netlist*. The algorithm has a single parameter named *maxout* with type *int*. This parameter signifies the maximum fanout for a *NetlistNode*. The default value for this parameter is *4*. The algorithm will duplicate the *NetlistNode* that exceeds the *maxout* limit, and *NetlistEdges* going to the *NetlistNode*. After the optimization, the functionality of the circuit remains unchanged.

4.3 Clustering

Clustering is the technique of grouping instances of *NetlistNode* into a larger *NetlistNode*. Figure 4.1 shows an example *Netlist*. There are nine *NetlistNodes*, labeled *0* through *8*. The connections between the *NetlistNodes* are the *NetlistEdges*. Figure 4.2 shows a potential clustering (grouping) solution for the *Netlist* in Figure 4.1. The clusters (groupings) are encapsulated by the dark green boxes. There are four clusters: 1) containing *NetlistNodes* with label 0,1,2 and 4, 2) containing *NetlistNode* 3, 3) containing *NetlistNode* 5, and, 4) containing *NetlistNode* 6,7 and 8. Figure 4.3 shows each clusters in Figure 4.2 represented as a *NetlistNode*. There are four *NetlistNodes*: one for each cluster in Figure 4.2.

The *ResultNetlistNodeData* class is modified with a cluster*ID* attribute and the methods for accessing this attribute. The *setClusterID(*cluster*ID)* method sets the value of the cluster*ID* attribute, and, the g*etClusterID()* method gets the value of the cluster*ID* attribute.

The cluster*ID* is of integer type, where the value represents the assigned cluster for the instance of the *NetlistNode* class. A negative number signifies that the instance of the *NetlistNode* class is not assigned to a cluster. A non-negative number signifies that the instance of the *NetlistNode* class is assigned to a cluster, the value corresponds to the *cluster*. Hence, instances of the *NetlistNode* class with an equivalent cluster*ID* value are assigned to the same *cluster*. For example, instances of the *NetlistNode* class with a cluster*ID* value set to *0* are assigned to the same *cluster*. For reference, a default value for an instance of a Netlist class that is not assigned to a *cluster* is defined in the *CLResult* class of the *results.clustering* package.

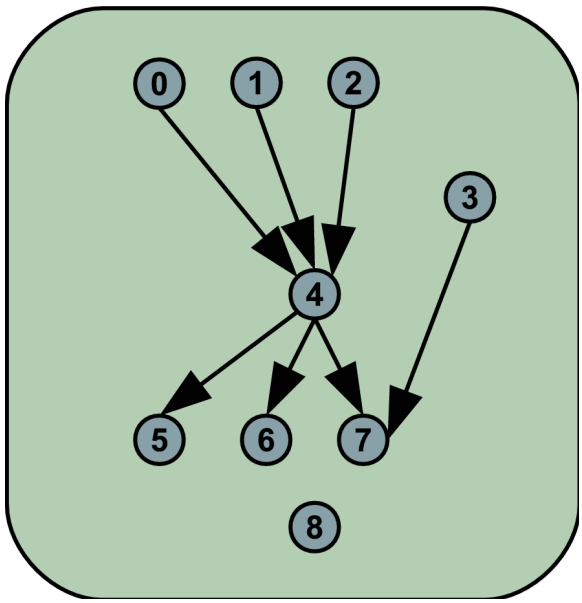


Figure 4.1: Example *Netlist*

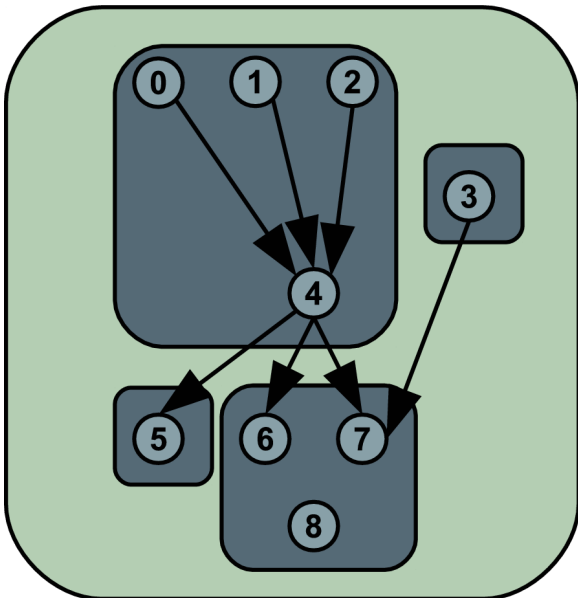


Figure 4.2: Potential clustering (grouping) for the Netlist in Figure 4.1

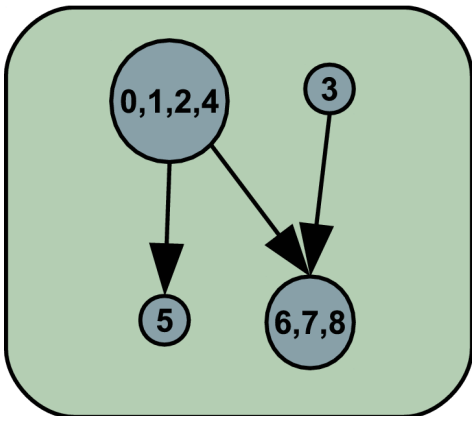


Figure 4.3: Each clusters in Figure 4.2 represented as a *NetlistNode*

4.3.1 CL_RC

The CL_RC is an algorithm that clusters instances of *NetlistNode* using resource and connection constraints. This algorithm is based on Texas University's algorithm from the mid 90's and is not implemented. For details of the algorithm: see notes.

4.4 Partitioning

The purpose of the *partitioning* stage is to assign the instances of the *NetlistNode* class of the Netlist to a block of a partition, where a block represents a biological cell. This stage contains algorithms implementing a solution for the *partitioning* problem. The problem is described in the publication found at:
./Cello/Documentation/DNACompiler/partitioning/ProblemDefinition/AnEfficientHeuristicProcedureForPartitioningGraphs.pdf.

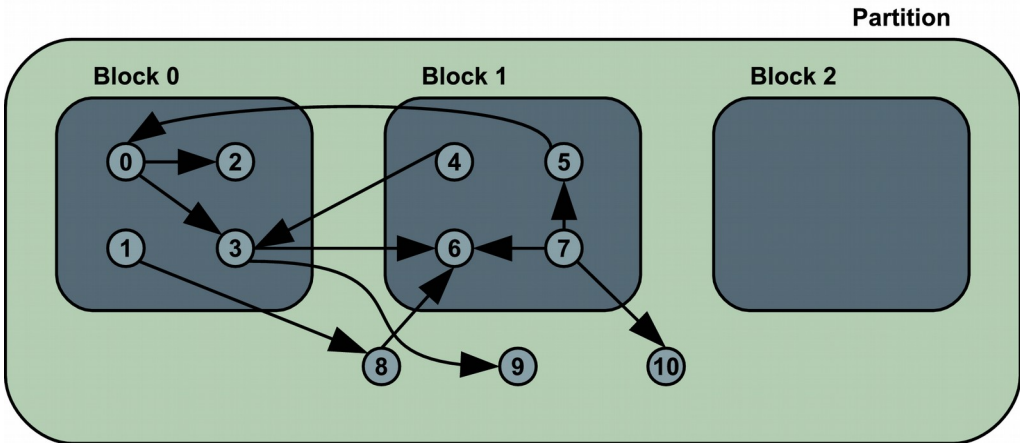


Figure 4.4: Example of a partial solution for the partitioning problem

Figure 4.4 shows an example of a partial solution for the partitioning problem. In the Figure, the partition contains three blocks: *Block 0*, *Block 1* and *Block 2*. Also, in the figure, there are 11 nodes labeled *0* through *10*. *Block 0* and *Block 1* contain four nodes, and, *Block 2* does not contain any nodes. There are three nodes that are not assigned to a block.

The *ResultNetlistNodeData* class is modified with a *partitionID* attribute and the methods for accessing this attribute. The *setPartitionID(partitionID)* method sets the value of the *partitionID* attribute, and, the *getPartitionID()* method gets the value of the *partitionID* attribute.

The *partitionID* is of integer type, where the value represents the assigned block for the instance of the *NetlistNode* class. A negative number signifies that the instance of the *NetlistNode* class is not assigned to a block. A non-negative number signifies that the instance of the *NetlistNode* class is assigned to a block, the value corresponds to the *block*. Hence, instances of the *NetlistNode* class with an equivalent *partitionID* value are assigned to the same block. For example, instances of the *NetlistNode* class with a *partitionID* value set to *0* are assigned to the same block. For reference, a default value for an instance of a Netlist class that is not assigned to a block is defined in the *PTResult* class of the *results.partitioning* package.

In Cello, the partition is described by a *profile*. The profile contains: 1) the number of blocks in the partition, and, 2) identifiers for the partition and the blocks. The profile is stored in the TargetData file, described in Chapter 13 of the Poros documentation.

```
{
  "collection": "PartitionProfile",
  "PartitionProfile":
        {
                "name": "PProfile0",
                "Blocks":
                {
                        "Capacity_Units":
                        [
                                "NOT",
                                "NOR",
                                "OR"
                        ],
                        "Capacity":
                        [
                                {
                                        "name": "capacity0",
                                        "units":
                                        [
                                                "NOT"
                                        ],
                                        "lower_bound": 0,
                                        "lower_bound_type": "greater_than_or_equal",
                                        "upper_bound": 3,
                                        "upper_bound_type": "less_than"
                                },
                                {
                                        "name": "capacity1",
                                        "units":
                                        [
                                                "NOR"
                                        ],
                                        "lower_bound": 0,
                                        "lower_bound_type": "greater_than_or_equal",
                                        "upper_bound": 6,
                                        "upper_bound_type": "less_than"
                                },
                                {
                                        "name": "capacity2",
                                        "units":
                                        [
                                                "OR"
                                        ],
                                        "lower_bound": 0,
                                        "lower_bound_type": "greater_than_or_equal",
                                        "upper_bound": 4,
```

```
                        "upper_bound_type": "less_than"
                }
        ],
        "Blocks":
        [
                {
                        "name" : "Block 0",
                        "capacity":
                        [
                                "capacity0",
                                "capacity1",
                                "capacity2"
                        ]
                },
                {
                        "name" : "Block 1",
                        "capacity":
                        [
                                "capacity0",
                                "capacity2"
                        ]
                },
                {
                        "name" : "Block 2",
                        "capacity":
                        [
                                "capacity0",
                        ]
                }
        ]
},
"InterBlocks":
{
        "Capacity_Units":
        [
                "Regular"
        ],
        "Capacity":
        [
                {
                        "name": "capacity0",
                        "units":
                        [
                                "Regular"
                        ],
                        "lower_bound": 0,
```

```
                                        "lower_bound_type": "greater_than_or_equal",
                                        "upper_bound": 4,
                                        "upper_bound_type": "less_than"
                                }
                        ],
                        "InterBlocks":
                        [
                                {
                                        "name" : "Interblock_0_1",
                                        "source" : "block0",
                                        "destination" : "block1",
                                        "capacity":
                                        [
                                                "capacity0"
                                        ]
                                },
                                {
                                        "name" : "Interblock_1_0",
                                        "source" : "block1",
                                        "destination" : "block0",
                                        "capacity":
                                        [
                                                "capacity0"
                                        ]
                                }
                        ]
                }
        }
},
```

Figure 4.5: The profile describing the partition illustrated in Figure 4.4

Figure 4.5 shows the profile describing the partition illustrated in Figure 4.4. The profile identifies the partition with name *PProfile0* that contains three blocks identified by *Block 0*, *Block 1* and *Block 2*. There are three capacity units for the blocks: *NOT*, *NOR* and *OR*. There are three types of capacity for the blocks: *capacity0*, *capacity1* and *capacity2*. Capacity *capacity0*, *capacity1* and *capacity2* have constraints $0 <= NOT < 3$, $0 <= NOR < 6$ and $0 <= OR < 4$ respectively.

An Interblock refers to the edges between blocks. For the interblocks of profile, there is one capacity unit for the edges between blocks, named *Regular*, and one capacity: *capacity0*. Capacity *capacity0*,has constraint $0 <= Regular < 4$. The latter constraint is applied between *Block 0* and *Block 1, and, Block 1* and *Block 0.*

4.4.1 HMetis

The HMetis algorithm uses a third party tool entitled hMETIS[10] - Hypergraph & Circuit Partitioning. Additional details on hMETIS can be found in the *./Cello/Documentation/DNACompiler/parititoning/HMetis* folder. The remainder of this section describes the HMetis algorithm.

There are no parameters for the HMetis algorithm. The HMetis algorithm executes the hMETIS tool. For the execution of the tool, the hMETIS tool requires values defined for its arguments. A description of the arguments are found in the hMETIS documentation.

Argument Name	**Value**
UBfactor	*1*
Nruns	*10*
CType	*1*
RType	*1*
Vcycle	*3*
Reconst	*0*
dbglvl	*0*

Table 5: List of arguments and their values when executing the hMETIS tool in the HMetis algorithm

The value of the argument corresponding to the number of blocks in the partition solution is retrieved from the partition profile defined in the Target Data file. Table 5 shows a list of arguments and their values when executing the hMETIS tool in the HMetis algorithm. The first column is the name of the argument, and, the second column is the assigned value.

4.4.2 GPCC_BASE

GPCC_BASE is an algorithm that performs analysis on the circuit to reduce the gate-to-block assignments. This algorithm contains the data structures and analysis procedures to be used by algorithms inheriting from this class, such as: *GPCC_SUGARM_BASE* and *GPCC_SCIP_BASE*.

Note that there is no partitioning algorithm associated with GPCC_BASE.

4.4.3 GPCC_SUGARM_BASE

The *GPCC_SUGARM_BASE* was implemented using a constraints programming paradigm. In the constraints programming paradigm, a set of variables and the relationship (constraints) between the variables are defined. Then, a solver is applied to derive the values for the variables that satisfy the constraints.

For the partitioning problem, the variables correspond to the possible gate-to-block assignments, and, the constraints refer to the target environment constraints with an additional constraint that a gate must be assigned to at most one block. The gate-to-block assignment variable is a binary value, where *true* denotes the cell is assign to a block, and *false* denotes the cell is not assign to a block.

10 http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview

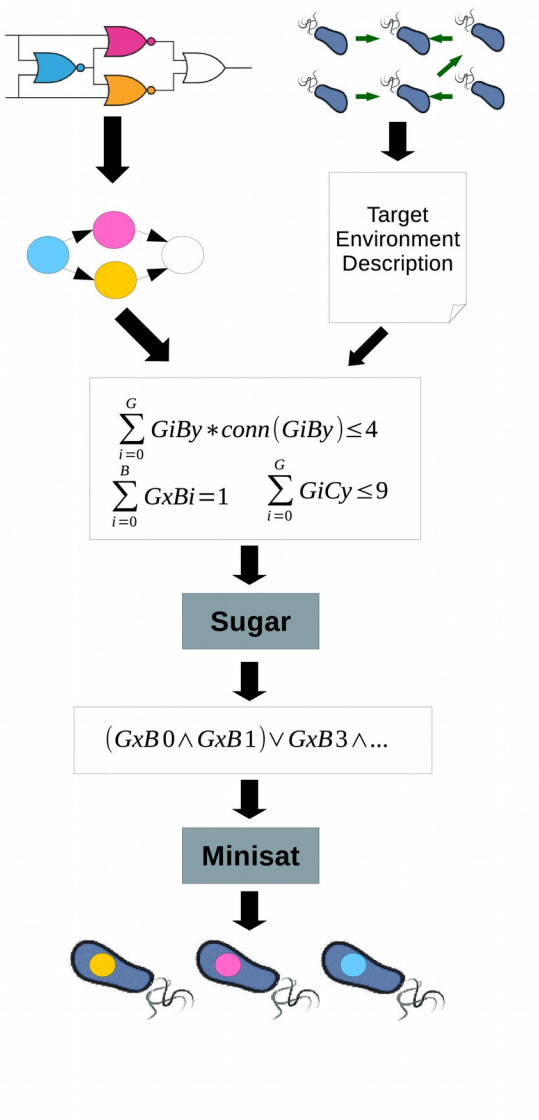


Figure 4.6: The implementation for the GPCC_SUGARM_BASE algorithm

Figure 4.6 illustrates the implementation of the GPCC_SUGARM_BASE algorithm. The circuit is translated into a graph, and the target environment is described using the markup language in Figure 4.5. The circuit and the target environment is used to generate the set of variables and constraints. The variables and constraints are passed to Sugar[11] that encodes these description into a boolean constraint satisfiability problem. The encoded data is the input to minisat[12], a solver for boolean satisfiability problems. If a solution exists, the output of minisat is the variables with their respective assigned values. Additional documentation for Sugar and minisat can be found at: *./Cello/Documentation/DNACompiler/partitioning/GPCC_SUGARM_BASE/Compiling finite linear*

11 http://bach.istc.kobe-u.ac.jp/sugar/
12 http://minisat.se/MiniSat.html

CSP into SAT.pdf, and,
./Cello/Documentation/DNACompiler/partitioning/GPCC_SUGARM_BASE/Minisat.pdf respectively.

4.4.4 GPCC_SCIP_BASE (Functionality Deprecated)

The GPCC_SCIP_BASE algorithm uses a third party tool: SCIP[13]. SCIP is currently one of the fastest non-commercial solvers for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP). It is also a framework for constraint integer programming and branch-cut-and-price. It allows for total control of the solution process and the access of detailed information down to the guts of the solver.

The GPCC_SCIP_BASE algorithm is designed to reduce the number of communication channels between the cells. The algorithm builds the constraints using the markup language in Figure 4.5. The output of the algorithm is the assignment of a *NetlistNode* to a *block*.

4.5 Placing

The placement problem is defined as placing a component (e.g. a block or a cell) into a location. For the placement problem, the target is a spatial device with a predefined layout. Additional information on the placement problem is described in:
./Cello/Documentation/DNACompiler/placing/ProblemDefinition/Problem.pdf.

The purpose of the *placing* stage is to find a location for the instance of a *NetlistNode* of a *Netlist*. The location is assigned to *placement* attribute of the instances of the *ResultNetlistNodeData* of a *NetlistNode* class of the *Netlist*.

The ResultNetlistNodeData class is modified with a *placement* attribute and the methods for accessing this attribute. The *setPlacement(placement)* method sets the value of the *placement* attribute, and, the g*etPlacement()* method gets the value of the *placement* attribute. The *placement* attribute is of String type.

4.5.1 GPCC_GRID

The GPCC_GRID is an algorithm to place components in a spatial device with a grid-like layout. This algorithm is based on Betz et al. and is not implemented. For details of the algorithm: see notes.

4.6 TechnologyMapping

The technologyMapping problem is defined as projecting the *Netlist, NetlistNode and NetlistEdge* to the target technology by mapping the *Netlist, NetlistNode and NetlistEdge* to primitives of the technology. In Cello, the target technology is a biological cell, and a *NetlistNode* is mapped to biological gate for a target cell type.

The purpose of the *technologyMapping* stage is to associate the *Netlist* representation to the primitives of a target technology. The primitives of the target technology are biological gate types. The assignment is stored in the *gateType* attribute of the instances of the *ResultNetlistNodeData* of a *NetlistNode* class of the *Netlist*.

The ResultNetlistNodeData class is modified with a *gateType* attribute and the methods for accessing this attribute. The *setGateType(gateType)* method sets the value of the *gateType* attribute, and, the g*etGateType ()* method gets the value of the *gateType* attribute. The *gateType* attribute is of String type.

13 http://scip.zib.de/

4.6.1 Cello_JY_TP

The *Cello_JY_TP* algorithm assign a gate (repressor + promoters) to a *NetlistNode*. The assignment is performed using Jonghyeon's (shinx097@mit.edu) tandem promoter model that is built on the original Cello repressor assignment mechanism[14]. The representation of the biological gate data is identical as presented in the previous version of Cello[15].


```
{
        "collection": "inducers",
        "name": "pBAD",
        "off_reu": 0.058912976,
        "on_reu": 8.185745477,
        "K": 0.13357037,
        "n": 4.840922018,
        "a": 0.050827951,
        "b": 0.632094139,
        "seq":
"ACTTTTCATACTCCCGCCATTCAGAGAAGAAACCAATTGTCCATATTGCATCAGACATT
GCCGTCACTGCGTCTT..."
}
```


Figure 4.7: an example for the *inducers* collection

For the *Cello_JY_TP* algorithm, the target data file was extended with three additional collections: 1) the *inducers* collection for *Primary Input* from the *logicSysthesis* stage, 2) the *output* collection for *Primary Output* from the *logicSysthesis* stage, and, 3) quorum signal for multi-cell (partitioned – Chapter 4.4) circuits.

Figure 4.7 shows an example for the *inducers* collection. Each entry of an inducer collection consists of: a name, off_reu, on_reu, K, n, a, b and DNA sequence. The name is used for reference; the off_reu and on_reu is used for gate assignment; the K, n, a, and b values are used in Jonghyeon's (shinx097@mit.edu) tandem promoter model for gate assignment; and, the DNA sequence is used when building the DNA sequence output of Cello.


```
{
        "collection": "output",
        "name": "YFP",
        "seq":
"CTGAAGCTGTCACCGGATGTGCTTTCCGGTCTGATGAGTCCGTGAGGACGAAACAGC
CTCTACAAATAATTTTGT..."
},
```


Figure 4.8: an example for the *output* collection

14 A.A.K. Nielsen et al., Science 352, aac7341 (2016). DOI: 10.1126/science.aac7341
15 http://cellocad.org/

Figure 4.8 shows an example for the *output* collection. Each entry of an output collection consists of: a name, and DNA sequence. The name is used for reference; and, the DNA sequence is used when building the DNA sequence output of Cello.


```
{
        "collection": "quorum_pair",
        "name": "NONCE_Q_Q0",
        "input" :{
                "name": "NONCE_Q_Q0",
                "off_reu": 0.013412352,
                "on_reu": 3.976450405,
                "K": 0.042456227,
                "n": 1.33128967,
                "a": 0.963695611,
                "b": 1.0,
                "seq":
"TATATATATATATATAATAGCTTCTTACCGGACCTGTAGGATCGTACAGGTTTACGCAAG
AAAATGGTTTGTTACTTTCGAATAAA"
        },
        "output" :{
                "name": "NONCE_Q_Q0",
                "seq":
"GAAGCTGTCACCGGATGTGCTTTCCGGTCTGATGAGTCCGTGAGGACGAAACAGCCT
CTACAAATAATTTTGTTTAATACTAGAGAAAGAGGGG..."
        },
}
```


Figure 4.9: an example for the *quorum_pair* collection

Figure 4.9 shows an example for the *quorum_pair* collection. Each entry of an *quorum_pair* collection consists of two entries: 1) input referring to quorum sensors and 2) output referring to quorum inducers. The input consists of: a name, off_reu, on_reu, K, n, a, b and DNA sequence. The name is used for reference; the off_reu and on_reu is used for gate assignment; the K, n, a, and b values are used in Jonghyeon's (shinx097@mit.edu) tandem promoter model for gate assignment; and, the DNA sequence is used when building the DNA sequence output of Cello. The output consist of: a name, and DNA sequence. The name is used for reference; and, the DNA sequence is used when building the DNA sequence output of Cello.

The algorithm assigns quorum pairs using a round-robin mechanism and ensures that a either the sensor or inducer of the quorum pair is assigned to a cell (block from the perspective of partitioning).

The algorithm assigned the *GateType* attribute with promoters separated with colon followed by the repressor. For example, the assignment *pBAD:pLux:P2_PhlF* refers to an implementation with promoters *pBAD* and *pLux*, and, the gate implementation *P2_PhlF.*

5 Tools

This chapter describes the tools in the Cello framework.

5.1 run.py

run.py is a python script that executes an executable from the Cello project. The tool will search the existing directory tree for loadable libraries.

For Linux-based systems, the tool searches from the directory that the tool was executed for **.so* and **.so.** files in directories containing **/linuxARCH/** and set the *LD_LIBRARY_PATH* variable. For MAC-based systems, the tool searches from the directory that the tool was executed for **.dylib* files in directories containing **/macARCH/** and set the *DYLD_LIBRARY_PATH* variable. For Windows-based systems, the tool searches from the directory that the tool was executed for **.dll* files in directories containing **/winARCH/** and set the *PATH* variable. The *ARCH* corresponds to architecture of the libraries. The values for ARCH should be *32* or *64* corresponding to 32-bit and 64-bit architectures respectively. For example: *./Cello/src/resources-partitioning/algorithms/GPCC_SUGARM_BASE/linux64* corresponds to a library path for a 64-bit linux-based system.

Moreover, the tool setups the classpath for the Java Virtual Machine (JVM). The tool searches from the directory that the tool was executed for **.jar* files in all subdirectories and sets the classpath variable (*-cp*). As a result, to add to the classpath, the *-classpath* argument must be used.

Note: depending on the system, the classpath may need to be edited to define the priority between Java archives (JAR).

5.1.1 Description

Help:

usage: run.py [-h] [-j JVM] -e EXECUTABLE [-a EXEC_ARGS]

-h, --help show this help message and exit

-j JVM, --jvm JVM Java Virtual Machine (VM) arguments

-e EXECUTABLE, --executable EXECUTABLE

Executable

-a EXEC_ARGS, --exec_args EXEC_ARGS

Executable Arguments

Execution signature:

run.py -j **JVM** -e **EXECUTABLE** -a **EXEC_ARGS**

-j JVM (Optional)

The JVM describes the Java virtual machine arguments. To add to the classpath, the *-classpath* argument must be used. Note that the list of arguments need to be surrounded with double quotes: e.g. “-Xms128M -XX:MaxNewSize:256m”.

-e EXECUTABLE (Required)

The EXECUTABLE describes the executable to execute. Refer to Chapter 8 of the Poros Documentation.

-a EXEC_ARGS (Optional)

The EXEC_ARGS describes the arguments for the executable. Note that the list of arguments need to be surrounded with double quotes: e.g. "-outputDir ../output/".

5.1.2 Examples

1) ./run.py -e DNACompiler
   Description: Execute the DNACompiler Executable.

2) ./run.py -e DNACompiler -j "-Xmx5G"
   Description: Execute the DNACompiler Executable with a maximum heap size of 5 GB.

3) ./run.py -e DNACompiler -j "-Xms2G -Xmx5G"
   Description: Execute the DNACompiler Executable with a minimum heap size of 2 GB and a maximum heap size of 5GB (a heap size between 2GB and 5GB).

4) ./run.py -e DNACompiler -j "-Xms2G -Xmx5G" -a "-inputNetlist adder.v -targetDataFile Eco1C1G1T1.UCF.json -options options.csv -netlistConstraintFile adder_netlistconstraints.json -outputDir ../sample-output/adder"
   Description: Execute the DNACompiler Executable with a minimum heap size of 2 GB and a maximum heap size of 5GB (a heap size between 2GB and 5GB). The executable is executed with: 1) an input netlist defined in the verilog file *adder.v*, 2) a target data defined in file *Eco1C1G1T1.UCF.json*, 3) options defined in file *options.csv*, 4) netlist contrainsts defined in file *adder_netlistconstraints.json*, and, 5) the output directory defined as *../sample-output/adder*.

5) ./run.py -e logicSynthesis -a "-inputNetlist adder.v -targetDataFile Eco1C1G1T1.UCF.json -options options.csv -netlistConstraintFile adder_netlistconstraints.json -outputDir ../sample-output/adder -algoName Yosys"
   Description: Execute the logicSynthesis Executable with the Yosys algorithm. The executable is executed with: 1) an input netlist defined in the verilog file *adder.v*, 2) a target data defined in file *Eco1C1G1T1.UCF.json*, 3) options defined in file *options.csv*, 4) netlist contrainsts defined in file *adder_netlistconstraints.json*, 5) the output directory defined as *../sample-output/adder*, and, 6) the *Yosys* algorithm.

5.2 PartitionProfileGenerator.py

PartitionProfileGenerator.py is a python script that generates the partition profile described in Chapter 4.4. The tool will generate the profile for a 2 dimensional layout spatial device. The profile is used to model the spatial device. The generator support two topologies: 1) 2-D Tree, and, 2) pipeline.

5.2.1 Description

Help:

usage: run.py [-h] -r NUMROWS -c NUMCOLUMNS -t TOPOLOGY

-h, --help show this help message and exit

-r NUMROWS, --numRows NUMROWS

Number of Rows

-c NUMCOLUMNS, --numColumns NUMCOLUMNS

Number of Columns

-t TOPOLOGY, --topology TOPOLOGY

Topology

Execution signature:

run.py -r **NUMROWS** -c **NUMCOLUMNS** -t **TOPOLOGY**

-r NUMROWS (Required)

The NUMROWS describes the number of rows for the 2 dimensional spacial device. When using the pipeline topology, the *NUMROWS* must be set to *1*.

-c NUMCOLUMNS (Required)

The NUMCOLUMNS describes the number of columns for the 2 dimensional spacial device.

-t TOPOLOGY (Required)

The TOPOLOGY describes the topology to model the spatial device. When using the pipeline topology, the *NUMROWS* must be set to *1*.

5.2.2 Examples

1) ./*PartitionProfileGenerator*.py -r 4 -c 4 -t tree
   Description: Generates a profile with a 4 by 4 tree topology.
2) ./*PartitionProfileGenerator*.py -r 6 -c 12 -t tree

   Description: Generates a profile with a 6 by 12 tree topology.
3) ./*PartitionProfileGenerator*.py -r 1 -c 44 -t pipeline

   Description: Generates a profile with a 1 by 44 tree pipeline.

6 Debugging Notes In Eclipse

The algorithms may use external libraries and executables. To load a given library in Eclipse for debugging. In a debug configuration in Eclipse, the *LD_LIBRARY_PATH* environment variable needs to be set. To set this variable, in the "Run Configuration..." or "Debug Configuration..." dialog box:

1) select the *Environment* tab,

2)

   a) if the *LD_LIBRARY_PATH* variable is not present in the list, click *New…*, enter *LD_LIBRARY_PATH* in the *Name* field

   b) if the *LD_LIBRARY_PATH* variable is present in the list, highlight the entry, click *Select…*

3) enter the path to the libraries: e.g. *./Cello/src/resources-partitioning/algorithms/GPCC_SCIP/linux64:./Cello/src/resources-partitioning/algorithms/GPCC_SUGARM_BASE/linux64*

4) Click OK to accept