Poros

An Open-source Development And Execution Framework For Circuit Design Automation

Technical Report and Documentation

Vincent Mirian and Christopher A. Voigt

Massachusetts Institute of Technology

Revised: January 2018

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

Table of Contents

1 Introduction

Poros, named after the personification of expedience and contrivance in Greek classical literature, is an open-source framework to manage and control the development and execution of a software project for circuit design automation. The framework is a set of: source files, libraries and tools written in the Java and Python programming languages. The framework also has enhanced support for projects developed in Eclipse[1]. It provides infrastructure for:

1) adding or removing a stage (a class of algorithms),
2) adding, removing or extending an algorithm,
3) modifying the parameters of an algorithm,
4) adding or removing an application (also known as a 'program'),
5) executing an application,
6) executing a stage,
7) managing and parsing command line arguments,
8) logging,
9) controlling the default execution flow of an application using a human-readable external file, and,
10) generating documentation for the Application Programming Interface (API) of the application(s), stage(s), and, algorithm(s) using Javadoc[2].

The framework generates the necessary infrastructure (files) for implementing an application, a stage and an algorithm. The framework also integrates the application, stage or algorithm within the project. From a user's perspective, the framework removes the burden of *stitching* the algorithm into a project allowing the user to focus on the *more* important task of: 1) implementing a stage or an algorithm, 2) preparing/processing the data for an algorithm, and, 3) designing the execution flow of an application. The remainder of this documentation describes the setup for the framework, the structure of the framework, and the steps for using the various features in the framework.

1 https://www.eclipse.org/
2 https://en.wikipedia.org/wiki/Javadoc

2 Getting Started

It is strongly suggested using the Virtual Machine installation method. The Virtual Machine installation method is an *out-of-the-box* solution.

2.1 Installation

2.1.1 Virtual Machine

1) Download the latest VirtualBox for the desired operating system (OS) at: https://www.virtualbox.org/wiki/Downloads.

   Note: On Ubuntu 16.04 LTS, virtualbox can be installed using the following commands:

   1) sudo apt-get install virtualbox build-essential module-assistant
   2) openssl req -new -x509 -newkey rsa:2048 -keyout MOK.priv -outform DER -out MOK.der -nodes -days 36500 -subj "/CN=VirtualMachine/"
   3) sudo mokutil --import MOK.der (supply password for later use)
   4) Reboot
   5) (https://sourceware.org/systemtap/wiki/SecureBoot)
      1. Enroll MOK
      2. Continue
      3. Yes
      4. password from step 3)
      5. OK
   6) sudo modprobe vboxdrv
2) Download the Virtual Disk Image at: **(TODO)**
3) Install VirtualBox on the desired system.
4) Machine > New…
5) Select Virtual Disk image from step 2)
6) Start
7) User: Poros Password: letmein

2.1.2 Ubuntu 16.04

Eclipse:

1) sudo add-apt-repository ppa:ubuntu-desktop/ubuntu-make
2) sudo apt update
3) sudo apt install ubuntu-make

4) umake eclipse

git:

1) sudo apt-get install git-core

Continue with instructions in Section 2.1.4.

2.1.3 Windows or MAC OS X

Eclipse:

1) Download from http://www.eclipse.org/downloads/
2) Install on machine

git:

1) Download from https://git-scm.com/downloads or https://desktop.github.com/
2) Install on machine

python (framework tested with 2.7.14 and 3.6.4):

1) Download the appropriate installer at: https://www.python.org/downloads/mac-osx/ (MAC OSX) https://www.python.org/downloads/windows/ (Windows)

Continue with instructions in Section 2.1.4.

2.1.4 Get source from repository

1) git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY **(TODO)**

Continue with instructions in Section 2.1.5.

2.1.5 Setup Eclipse

Import Project into Eclipse

1) Open eclipse:

   Note: In Ubuntu terminal type: eclipse-java

2) Select workspace directory
3) File > Import
4) Existing Projects into Workspace
5) Navigate to directory where repository is cloned

Enable Ant build script:

1. Window > Show View > Ant
2. Right click on Project > Properties
3. Builders > Import > ...
4. Find and select "build.xml" under the Partitioner directory

Add External Jars to Project:

1. Click on Windows > Preferences
2. Java > Build Path > User Libraries > New...
3. Enter 'PorosCompilerlib'
4. Select 'PorosCompilerlib'
5. Click Add Jars...
6. Select all jars in ./external_jars

3 Overview

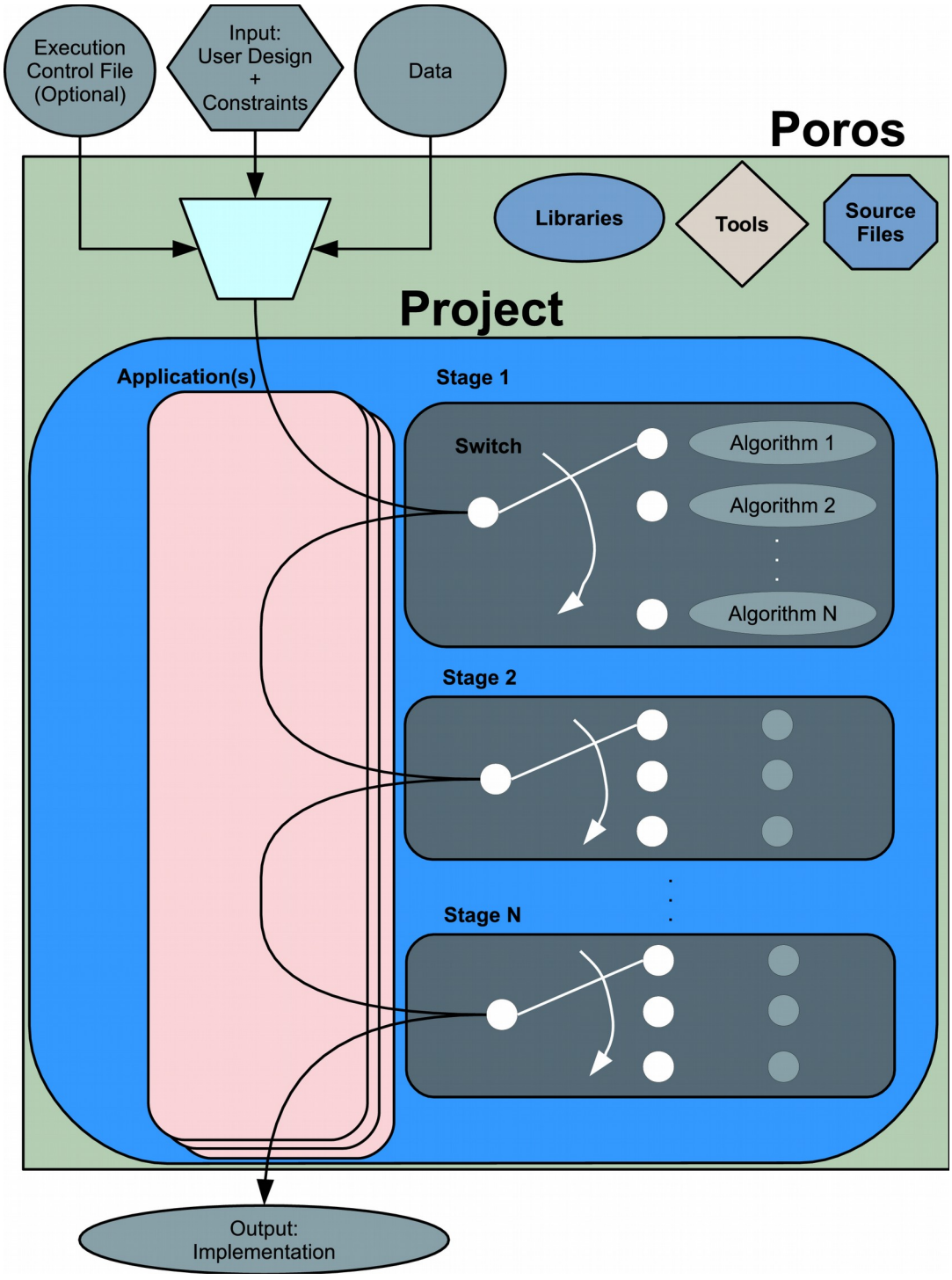

Figure 3.1: Overview of the Poros framework.

Figure 3.1 shows an overview of the framework. The framework is composed of source files, libraries and tools to: manage and control the development of a software project for circuit design automation. In figure 3.1, the framework is enclosed within the green box. The components of the framework are: 1) source files and libraries supporting the execution of an application and/or stage within a project, 2) tools for managing and controlling the development of a project and 3) a software project. Examples of tools include: 1) adding a stage into a project, and, 2) adding an application into a project.

The inputs to the framework are: 1) a description of the user's circuit design and design constraints, 2) data for deriving the implementation on the target, and, 3) an (optional) execution control file. The output of the framework is an implementation of the circuit design for the given target.

The project is composed of: 1) application(s), 2) stage(s) and 3) algorithm(s). The inputs are passed to each stage in an application to compute a result guiding the design towards an implementation. The application defines the order of execution for the stages (execution flow). A stage is a collection of algorithms, and, an algorithm is a procedure for calculating the result of a stage. In brief, an application executes stages, and stages contain algorithms.

The purpose of an application is to define the default execution flow for deriving an implementation for the user's circuit design. The purpose of a stage is to: 1) compute a result for an attribute of the user's circuit design, and, 2) select the algorithm for computing this result. The purpose of an algorithm is to compute a result for an attribute of the user's circuit design.

For each stage within a project, a default algorithm with default parameters is executed when the stage is executed. The algorithm selection and its default parameter values can be overwritten using the optional execution control file. The optional control file controls the switch at each stage to select the appropriate algorithm and/or override the default parameter values of the algorithm.

4 Poros Structure

This Chapter discusses the directory structure of the Poros framework.

4.1 Directory Structure

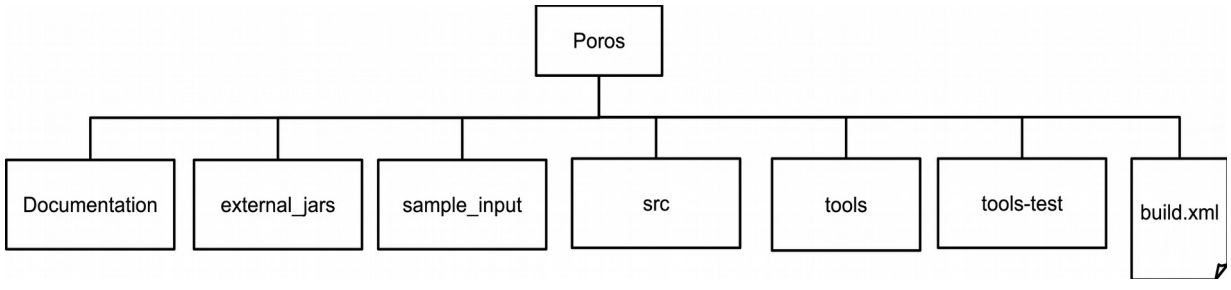

Figure 4.1: Top level directory structure of the Poros framework

Figure 4.1 illustrates the top level directory structure of the Poros framework. In the figure, folders are represented by rectangular boxes and files are represented by rectangular boxes with a folded bottom right corner.

The *Documentation* folder contains: 1) this document, and, 2) documentation for the Application Programming Interface (API) of the framework generated using Javadoc[1]. The *external_jars* folder contains the framework libraries (Chapter 5). The *sample_input* folder is a placeholder for sample inputs to an application and/or a stage. When an application or a stage is added to a project, a folder representing the application or stage is created within this *sample_input* folder. It is recommended to place test units for a given application or stage within their respective folders.

The *src* folder contains: 1) the source files of the framework and 2) project-specific applications, stages and algorithms. The *tools* folder contains the tools of the framework (Chapter 6). The *tools-test* folder contains a set of test units for the tools. The *build.xml* file is the build script for a project using the framework. **Note that is recommend not to modify the directory structure of the framework as unexpected behavior may occur such as: the build script not function correctly.**

1 https://en.wikipedia.org/wiki/Javadoc

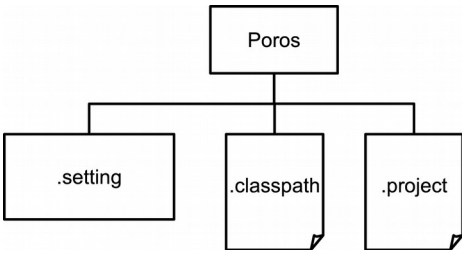

Figure 4.2: Components of the top level directory structure for an Eclipse Project

Figure 4.2 illustrates components of the top level directory structure for an Eclipse Project. Similar to Figure 4.1, folders are represented by rectangular boxes and files are represented by rectangular boxes with a folded bottom right corner.

The *.setting* folder contains settings for a project using the framework in Eclipse. The *.classpath* file contains the classpath information for a project using the framework in Eclipse. The *.project* file contains project-specific information for a project using the framework in Eclipse.

5 Libraries

The supporting libraries are written primarily in the JAVA programming language and are found in the *external_jars* folder of the project that uses the Poros framework. For consistency, all libraries written in Java (Java Archive - JAR) files used for the framework should be placed in this folder. When a new library is added to the framework, update the User Library as described in Section 2.1.5.

Filename	**Description**
ant-contrib.jar[1]	The Ant-Contrib project is a collection of tasks (and at one point maybe types and other tools) for Apache Ant.
commons-cli-1.4.jar[2]	The Apache Commons CLI library provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool.
commons-csv-1.5.jar[3]	Commons CSV reads and writes files in variations of the Comma Separated Value (CSV) format.
json-simple-1.1.1.jar	A simple Java toolkit for JSON.
log4j-1.2-api-2.10.0.jar[4]	Apache Log4j 2 is a Java toolkit for command line argument management and parsing.
log4j-api-2.10.0.jar[4]	
log4j-core-2.10.0.jar[4]	

Table 1: List of libraries in the framework

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

1 http://ant-contrib.sourceforge.net/
2 https://commons.apache.org/proper/commons-cli/
3 https://commons.apache.org/proper/commons-csv/
4 https://logging.apache.org/log4j/2.0/

6 Tools

This chapter describes the tools in the Poros framework. The tools require Python for execution. Instructions for installing Python on selected Operating Systems are found in Chapter 2.

6.1 instantiate_poros.py

instantiate_poros.py is a python script that instantiates the Poros framework for use by a new project.

6.1.1 Description

Help:

usage: instantiate_poros.py [-h] -p PROJECTDIR -f PORUSDIR -n PROJECTNAME [-s]

instantiate_poros.py

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

-p PROJECTDIR, --projectDir PROJECTDIR

Project Directory

-f PORUSDIR, --porusDir PORUSDIR

Poros Directory

-n PROJECTNAME, --projectName PROJECTNAME

Project Name

-s, --source Source only

Execution signature:

instantiate_poros.py -p **PROJECTDIR** -f **PORUSDIR** -n **PROJECTNAME [**-s]

-p PROJECTDIR (Required)

The PROJECTDIR describes the path of the project to instantiate the Poros framework. The path must not exist, it will be created.

-f PORUSDIR (Required)

The PORUSDIR describes the path to the root of the Poros framework.

-n PROJECTNAME (Required)

The PROJECTNAME describes the name of the project that will be using the Poros framework.

-s (Optional)

The -s is a switch. When the switch is enabled (present), the tool only instantiates the source files of the framework. Otherwise, it instantiates the complete infrastructure (source files, libraries and tools) of the framework.

6.1.2 Examples

1) ./instantiate_poros.py -p ../../Cello/ -f ../../Poros -n Cello
   Description: Instantiates the complete framework's infrastructure into the *Cello* project path. The project name is *Cello*.

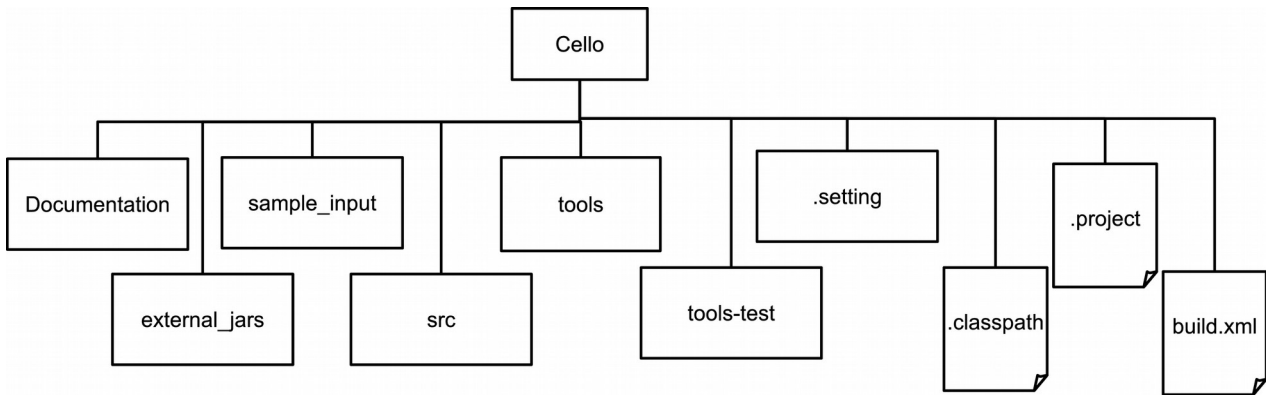

Figure 6.1: Top level directory structure of the *Cello* project after executing 1).

Figure 6.1 shows top level directory structure of the *Cello* project after executing 1).

2) ./instantiate_poros.py -p ../../Cello/ -f ../../Poros -n Cello -s
   Description: Instantiates the source files of the framework's infrastructure into the *Cello* project path.

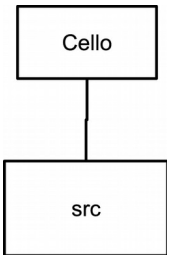

Figure 6.2: Top level directory structure of the *Cello* project after executing 2).

Figure 6.2 shows top level directory structure of the *Cello* project after executing 2).

6.2 add_remove_application.py

add_remove_application.py is a python script that adds or removes an application within a project that uses the Poros framework.

When an application is added to a project, three directories are added to the project: 1) a source directory containing the source files and infrastructure files for: a) developing the application, and, b) executing the application, 2) a resource directory containing external files needed during the execution of the application, and, 3) a directory for storing user-defined sample inputs for executing or testing the application, this directory is found within the *sample-input* directory of the framework. Note that executable external files must be placed in the *executable* folder of the resource directory. When an application is removed from a project, these directories are removed from the project.

In addition to adding directories, the tool generates: 1) application-specific source files and infrastructure files to aid in the development and execution the application, and, 2) a configuration file defining the default algorithm to execute when a stage is executed. When an application is removed from a project, these files are removed from the project.

6.2.1 Description

Help:

usage: add_remove_application.py [-h] -p PROJECTDIR -a APPNAME [-w AUTHORNAME] [-r]

add_remove_application.py

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

-p PROJECTDIR, --projectDir PROJECTDIR

Project Directory

-a APPNAME, --appName APPNAME

Application Name

-w AUTHORNAME, --authorName AUTHORNAME

Author(s) Name

-r, --remove Remove

Execution Signature:

add_remove_application.py -p **PROJECTDIR** -a **APPNAME** [-w **AUTHORNAME**] -r

-p PROJECTDIR (Required)

The PROJECTDIR describes the path to the root of the project that uses the Poros framework.

-a APPNAME (Required)

The APPNAME is the name of the application to be added to the project or removed from the project.

-w AUTHORNAME (Optional)

The AUTHORNAME is the name(s) of the author(s). For name(s) using spaces, enclose this field with double quotes(“”). Example: -w “Vincent Mirian and Christopher A. Voigt”. The default author is no author (an empty string - “”).

-r (Optional)

The -r is a switch. When the switch is enabled (present), the tool removes the application with APPNAME from the project. Otherwise, the tool adds an application with APPNAME to the project.

6.2.2 Example

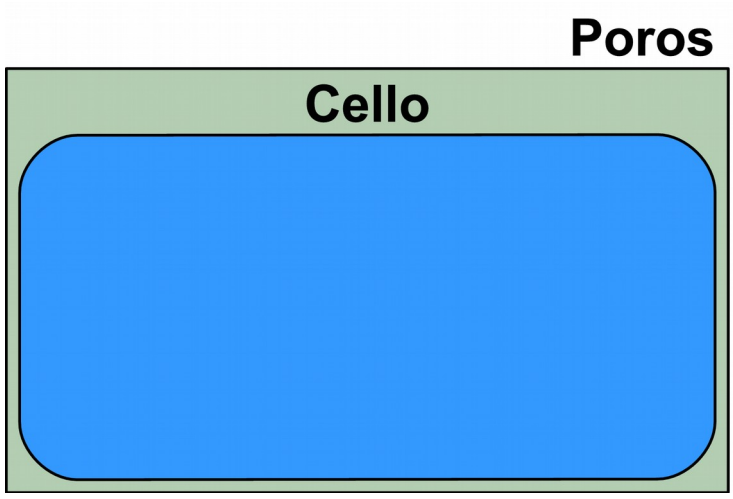

Figure 6.3: Initial state of the project using the framework

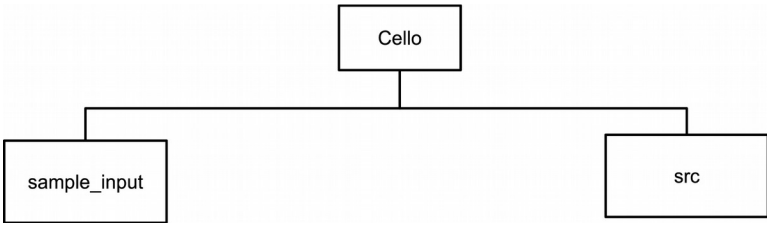

Figure 6.4: Initial state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the *Cello* project.

For the following examples, the initial state of the project using the framework is shown in Figure 6.3, and, the initial state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project is shown in Figure 6.4. The framework contains a project with name *Cello*.

1) ./add_remove_application.py -p ../../Cello/ -a DNACompiler

Description: Add the application 'DNACompiler' to the *Cello* project directory with no Author. The following directories are added to the project:

- *../../Cello/src/DNACompiler*
- *../../Cello/src/resources-DNACompiler*
- *../../Cello/src/resources-DNACompiler/executable*
- *../../Cello/sample-input/DNACompiler*

The generated application-specific source files for developing and executing the application are found in *../../Cello/src/DNACompiler*, and, the configuration file is found at *../../Cello/src/resources-DNACompiler/Configuration.json*.

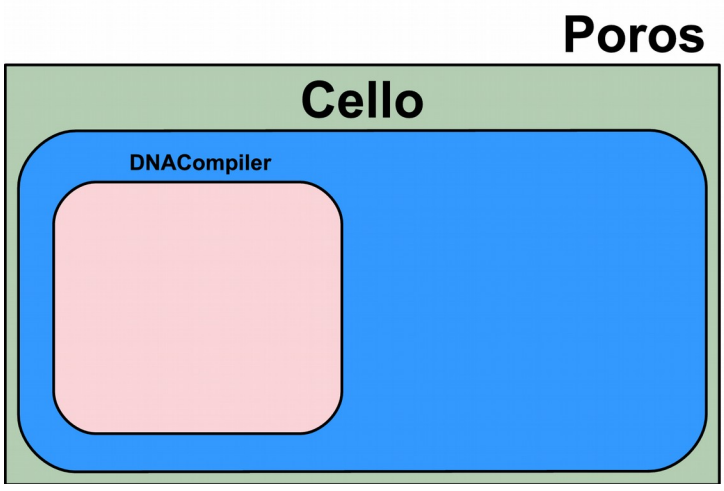

Figure 6.5: The state of the project after executing step 1).

Figure 6.5 shows the state of the project after executing step 1).

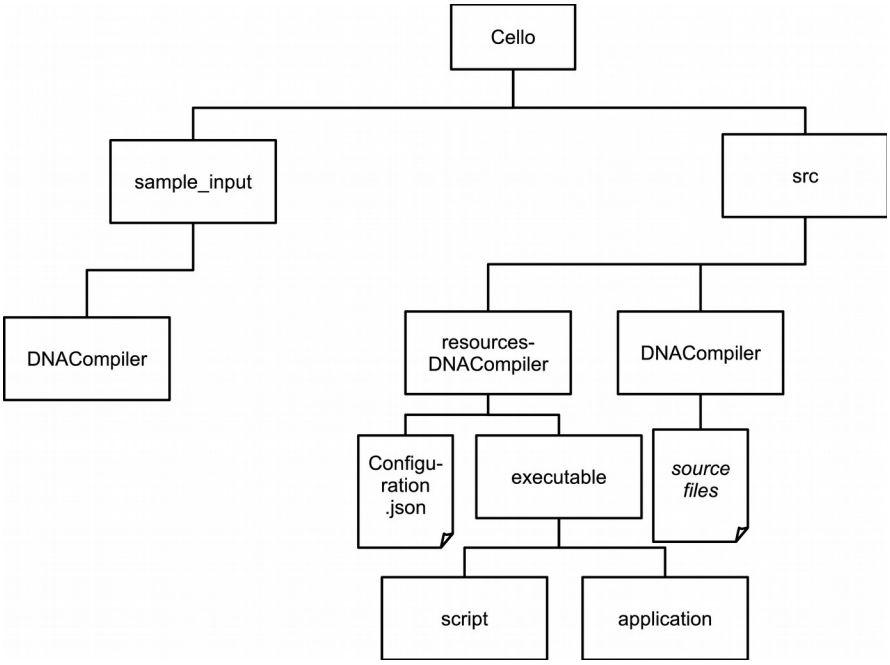

*Figure 6.6: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 1).

Figure 6.6 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 1).

2) ./add_remove_application.py -p ../../Cello/ -a other -w “John Doe”

Description: Add the application ‘other’ to the Cello project directory with Author “John Doe”.The following directories are added to the project:

- *../../Cello/src/other*
- *../../Cello/src/resources-other*
- *../../Cello/src/resources-other/executable*
- *../../Cello/sample-input/other*

The generated application-specific source files for developing and executing the application are found in *../../Cello/src/other*, and, the configuration file is found at *../../Cello/src/resources-other/Configuration.json*.

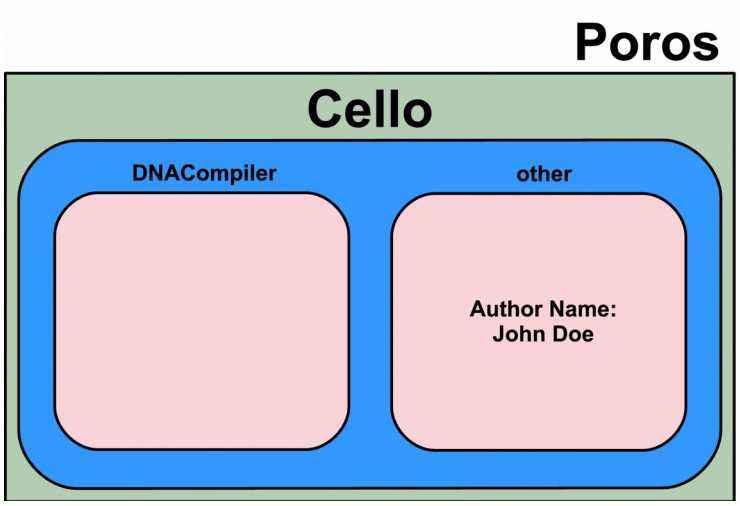

Figure 6.7: The state of the project after executing step 2).

Figure 6.7 shows the state of the project after executing step 2).

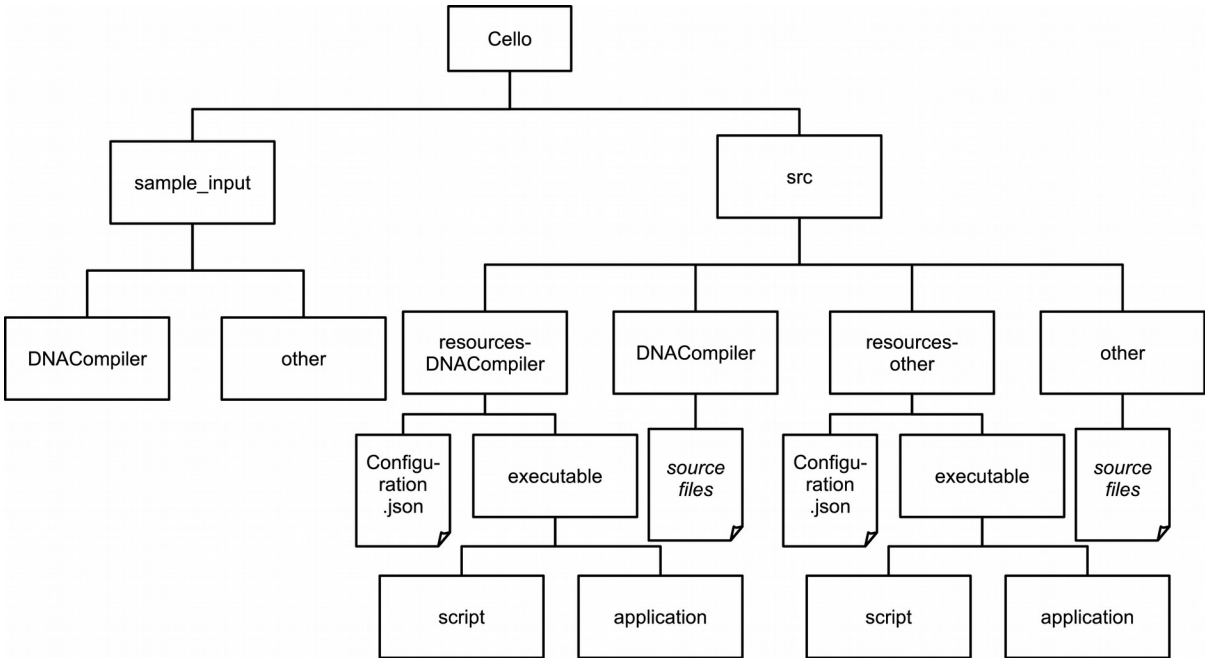

*Figure 6.8: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 2).

Figure 6.8 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 2).

3) ./ add_remove_application.py -p ../../Cello/ -a DNACompiler -r

Description: Remove the application ‘DNACompiler’ from the *Cello* project. The following directories are removed from the project:

- *../../Cello/src/DNACompiler*
- *../../Cello/src/resources-DNACompiler*
- *../../Cello/sample-input/DNACompiler*

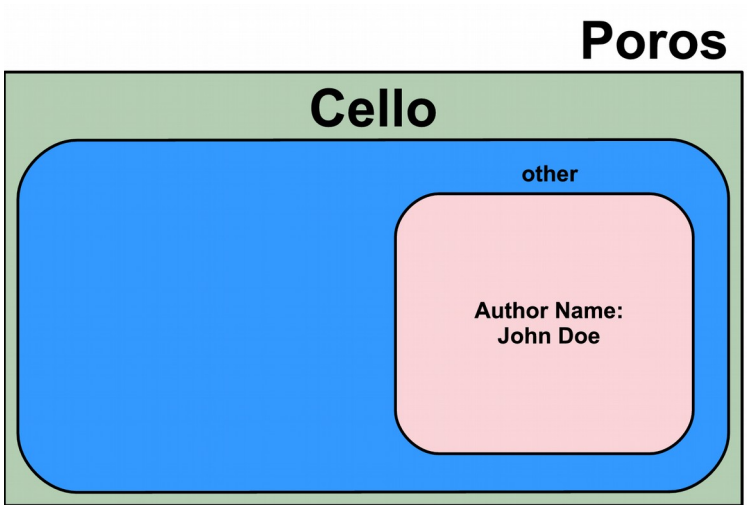

Figure 6.9: The state of the project after executing step 3).

Figure 6.9 shows the state of the project after executing step 3).

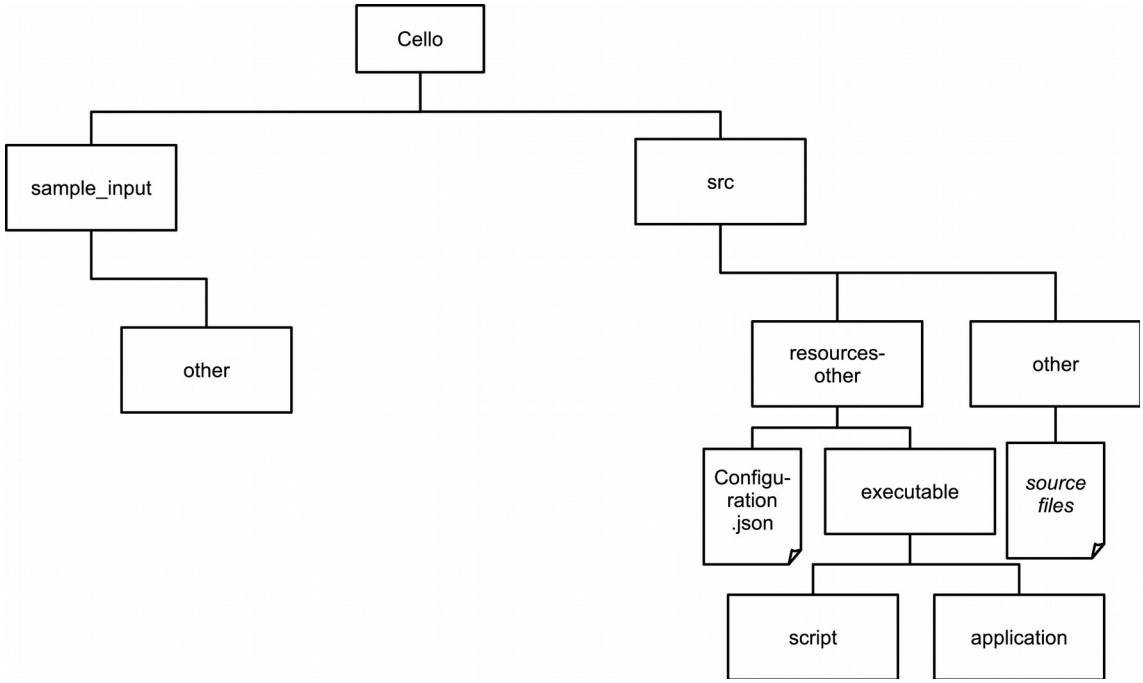

*Figure 6.10: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 3).

Figure 6.10 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 3).

6.3 add_stage_algo.py

add_stage_algo.py is a python script that adds a stage and/or an algorithm into a project that uses the Poros framework.

When a stage is added to a project, four directories are added to the project: 1) a source directory containing the source files and infrastructure files for executing the stage, 2) a resource directory containing external files needed during the execution of the stage, and, 3) a directory for storing user-defined sample inputs for executing or testing the stage, this directory is found within the *sample-input* directory of the framework. Note that executable external files must be placed in the *executable* folder of the resource directory. In addition to adding directories, the tools generates: 1) stage-specific source files and infrastructure files for executing the stage, 2) a JAVA package in the *results* folder for storing results from the stage, 3) a JAVA package in the *data* folder for storing target data for the stage, and, 4) a JAVA package in the *constraint* folder for storing constraints for the stage.

When an algorithm is added to a stage of a project, two directories are added to the project: 1) a directory containing the source files for the algorithm is added to the stage's source directory, and, 2) a directory containing algorithm-specific file(s), this directory is in the stage's resource directory. In addition to adding the directory, the tool modifies the stage's existing infrastructure files and generates algorithm-specific source files to aid in the development of the algorithm. Moreover, the tool generates: 1) a configuration file containing the algorithm parameters, their type and their default values, and, 2) a template file for extending the algorithm. These files are stored in the directory containing algorithm-specific file(s) located in the stage's resource directory.

6.3.1 Description

Help:

usage: add_stage_algo.py [-h] -i INFILE -p PROJECTDIR

add_stage_algo.py

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

-i INFILE, --infile INFILE

Input file

-p PROJECTDIR, --projectDir PROJECTDIR

Project Directory

Execution signature:

add_stage_algo.py -i **INFILE** -p **PROJECTDIR**

-i INFILE (Required)

The *INFILE* file contains the information for adding the infrastructure (files) for a stage and/or an algorithm within a project. It is in a comma-separated value (CSV) format.

AuthorName,*NewAuthor*,
ApplicationNames,*DNACompiler,other*
StagePrefix,*LS*,
StageName,*logicSynthesis*,
AlgorithmName,*NewBase*,
AlgorithmExtends,*Base*,
BooleanType,*TRUE*,boolean,
ByteType,*0*,byte,
CharType,*c*,char,
ShortType,*0*,short,
IntegerType,*0*,int,
LongType,*0*,long,
FloatType,*0.0*,float,
DoubleType,*0.0*,double,
StringType,"str",string,ADDED

Figure 6.11: Example of an *INFILE* file.

Figure 6.11 shows an example of an *INFILE* file. The first six lines of the file contain the required fields for adding a stage or an algorithm. The field names are in bold and the field values are in italic. The last nine lines of the file describe the parameters of an algorithm that can be controlled by the user from the optional execution control file. The parameter names are in bold, the parameter values are in italic, and the parameter types are underlined. The fourth field in a parameter entry annotates the existence of a parameter within an algorithm when an algorithm extends another. The keyword ADDED signifies that a parameter is new.

AuthorName: The value of this field describes the name of the author(s). The value is placed in the generated infrastructure (files) for an algorithm in Javadoc format. In the example file of Figure 6.11, the value of this field is *NewAuthor*.

ApplicationNames: The value of this field describes the name of the application to add the stages and/or algorithms. The values of this field are separated by a comma (,). In the example file of Figure 6.11, the values of this field are *DNACompiler* and *other*.

StagePrefix: The value of this field is an abbreviated string referencing the stage name. The abbreviation is prepended to the generated infrastructure files. The value is also used as an instance name in the generated infrastructure (files). This value must be unique within the project. In the example file of Figure 6.11, the value of this field is *LS*.

StageName: The name of the stage to be added to the framework. If the stage exists within the framework, then the algorithm will be inserted into the stage. If the stage does not exist, the stage will be generated and added into the project. This value must be unique within the project. In the example file of Figure 6.11, the value of this field is *logicSynthesis*.

AlgorithmName: The name of the algorithm to be added. This value must be unique within a stage of the project. If the stage associated with the algorithm does not exist, then the stage will be added to the project, and, the algorithm becomes the default algorithm for the stage of the applications listed in **ApplicationNames**. In the example file of Figure 6.11, the value of this field is *NewBase*.

AlgorithmExtends: The value of this field is the name of the algorithm that will be extended. No value present (an empty string) in the field signifies that the algorithm does not extend an existing algorithm within the stage, implying the algorithm is a standalone algorithm. In the example file of Figure 6.11, the value of this field is *Base*. Hence, the algorithm *NewBase* extends *Base*. Note this feature leverages the inheritance feature of the Java programming language.

Each entry of a parameter for an algorithm in the *INFILE* has the following format: **Name**, *Value*, Type, ADDED. **Name** represents an identifier for the parameter entry, *Value* corresponds to the default value of the parameter, Type defines the parameter type and ADDED annotates new parameters to be added for an algorithm that extends another. When extending an algorithm, only the parameters annotated with ADDED are added to the algorithm.

The name of a parameter can be any acceptable identifier in the Java programming language, the value of the parameter can be any acceptable value for the parameter type, and the type of the parameter can be any primitive type of Java programming language (e.g. boolean, byte, char, short, int, long, float, double) or a String. In the example file of Figure 6.11, **BooleanType** and **ByteType** are names of parameters (identifiers) of type boolean and byte respectively. Their default values are: *true* for **BooleanType** and *0* for **ByteType**. In the example file of Figure 6.11, the parameter with name **StringType** is associated with the algorithm *NewBase* and will be added to the *NewBase* algorithm.

-p PROJECTDIR (Required)

The PROJECTDIR describes the path to the root of the project that uses the Poros framework.

6.3.2 Example

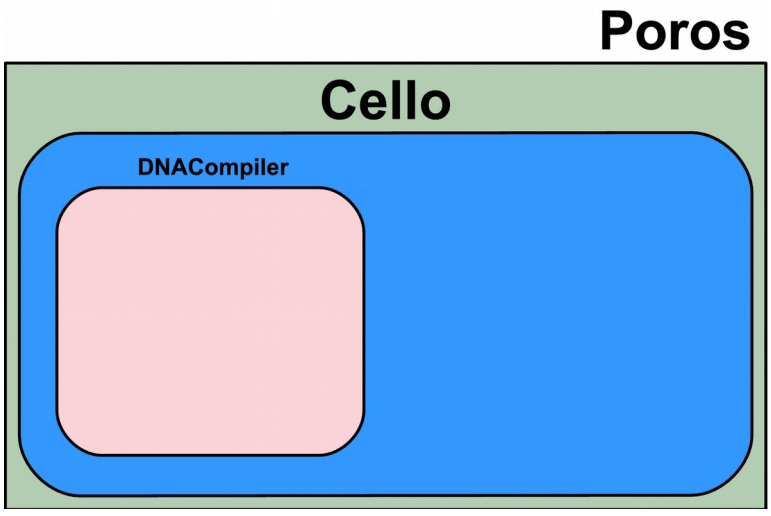

Figure 6.12: Initial state of the project using the framework

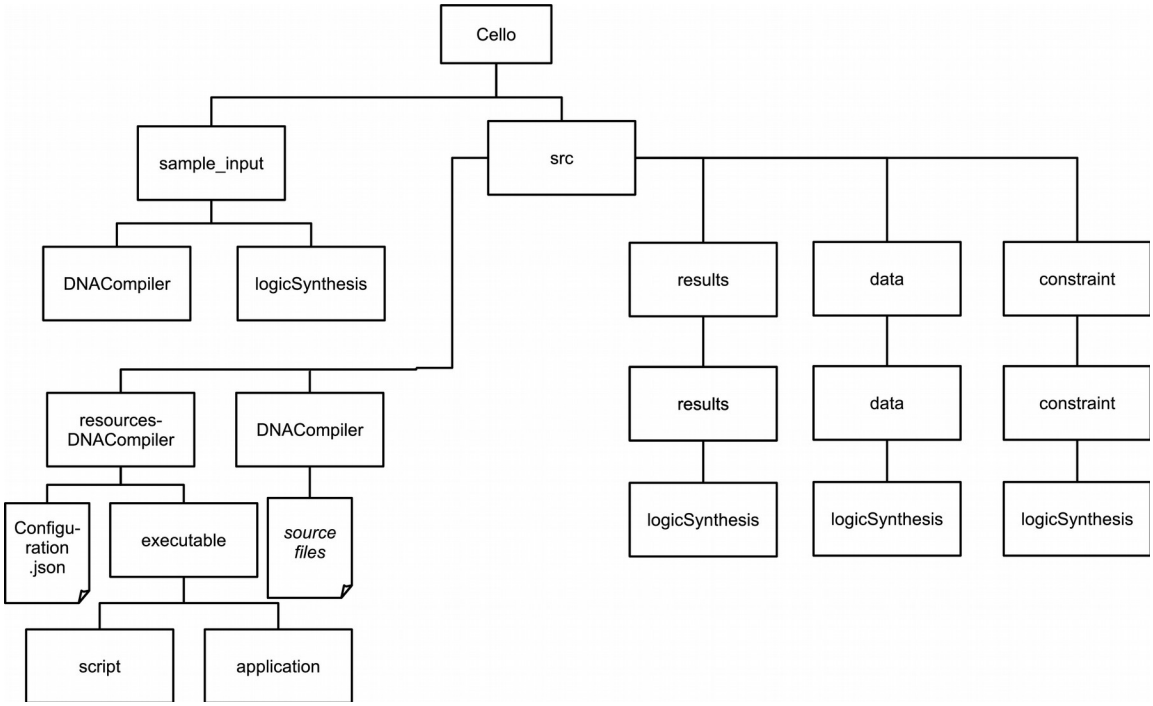

Figure 6.13: Initial state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the *Cello* project.

For the following examples, the initial state of the project using the framework is shown in Figure 6.12, and, the initial state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project is shown in Figure 6.13. The framework contains a project with name *Cello* that contains an application name *DNACompiler*. The corresponding directories of the *DNACompiler* application is found in the *sample-input* folder and the *src* folder.

1) ./add_stage_algo.py -i newStage.csv -p Cello/

AuthorName,*NewAuthor*,
ApplicationNames,*DNACompiler*,*other*
StagePrefix,*LS*,
StageName,*logicSynthesis*,
AlgorithmName,*Base*,
AlgorithmExtends,,
BooleanType,*TRUE*,boolean,
ByteType,*0*,byte,
CharType,*c*,char,
ShortType,*0*,short,
IntegerType,*0*,int,
LongType,*0*,long,
FloatType,*0.0*,float,
DoubleType,*0.0*,double,
StringType,*"str"*,string,

Figure 6.14: INFILE file (newStage.csv) for step 1).

Figure 6.14 shows the content of the *INFILE* file (newStage.csv) used in step 1).

Description: Given that the *logicSynthesis* stage is not present in the *Cello* project, the stage *logicSynthesis* is added to *Cello* project. In addition, the framework configures the *logicSynthesis* stage to have the default algorithm *Base* for applications:1) *DNACompiler* (by modifying the *Cello/src/resources-DNACompiler/Configuration.json* file) and 2) *other*. The following directories are added to the project:

- *Cello/src/logicSynthesis/logicSynthesis* (contains source files and infrastructure files for executing the stage)
- Cello/src/results/results/logicSynthesis (a JAVA package folder for storing results from the stage)
- Cello/src/data/data/logicSynthesis (a JAVA package folder for storing data for the stage)
- Cello/src/constraint/constraint/logicSynthesis (a JAVA package folder for storing constraints for the stage)
- *Cello/src/logicSynthesis/logicSynthesis/algorithm* (contains source files for the algorithms added to the stage)
- *Cello/src/resources-logicSynthesis* (contains external files)
- *Cello/src/resources-logicSynthesis/executable* (contains executable external files needed during the execution of the stage)
- *Cello/src/resources-logicSynthesis/algorithms* (contains the configuration file of algorithm(s) and the template file for extending algorithm(s))

- *Cello/sample-input/logicSynthesis* (contain user-defined sample inputs for executing or testing the stage)

For the *Base* algorithm, the generated algorithm-specific source files are found in *Cello/src/logicSynthesis/logicSynthesis/algorithm/Base*, its configuration file is found in *Cello/src/resources-logicSynthesis/algorithms/Base/Base.json*, and, its template file for extending the algorithm is found in *Cello/src/resources-logicSynthesis/algorithms/Base/Base.csv*. The *Cello/src/resources-logicSynthesis/algorithms/Base* directory is the directory for storing algorithm-specific files.

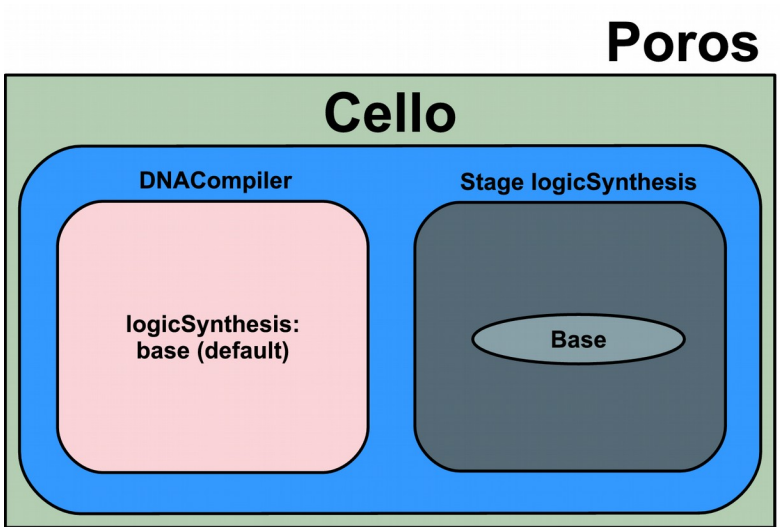

Figure 6.15: The state of the project after executing step 1).

Figure 6.15 shows the state of the project after executing step 1).

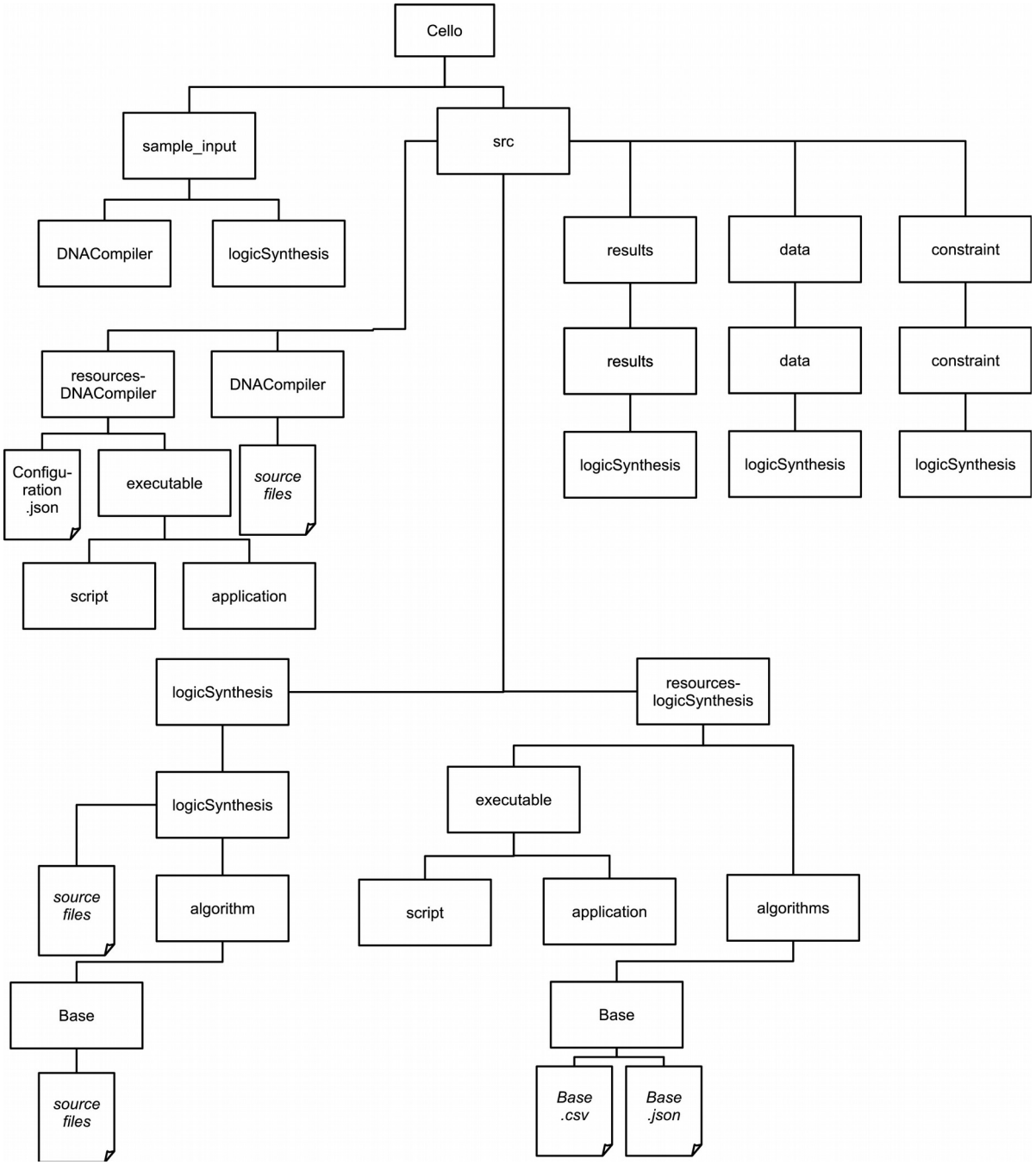

*Figure 6.16: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 1).

Figure 6.16 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 1).

2) ./add_stage_algo.py -i newAlgo.csv -p Cello/

AuthorName,*NewAuthor*,
ApplicationNames,*DNACompiler*,*other*
StagePrefix,*LS*,
StageName,*logicSynthesis*,
AlgorithmName,*NewBase*,
AlgorithmExtends,*Base*,
BooleanType,*TRUE*,boolean,
ByteType,*0*,byte,
CharType,*c*,char,
ShortType,*0*,short,
IntegerType,*0*,int,
LongType,*0*,long,
FloatType,*0.0*,float,
DoubleType,*0.0*,double,
StringType,*"str"*,string,

Figure 6.17: INFILE file (newAlgo.csv) for step 2).

Figure 6.17 shows the content of the *INFILE* file (newAlgo.csv) used in step 2).

Description: The algorithm *NewBase* is added to the *logicSynthesis* stage in the *Cello* project. For the *NewBase* algorithm, the generated algorithm-specific source files are found in *Cello/src/logicSynthesis/logicSynthesis/algorithm/NewBase*, its configuration file is found in *Cello/src/resources-logicSynthesis/algorithms/NewBase/NewBase.json*, and, its template file for extending the algorithm is found in *Cello/src/resources-logicSynthesis/algorithms/NewBase/NewBase.csv*. The *Cello/src/resources-logicSynthesis/algorithms/NewBase* directory is the directory for storing algorithm-specific files.

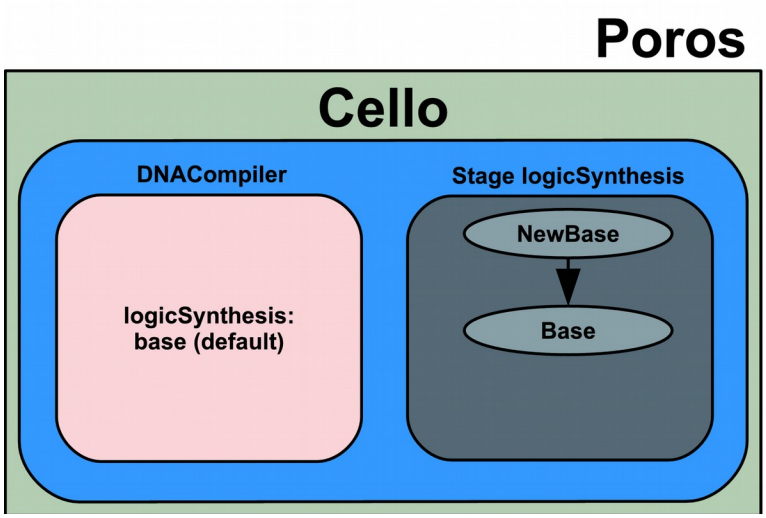

Figure 6.18: The state of the project after executing step 2).

Figure 6.18 shows the state of the project after executing step 2). The arrow between *NewBase* and *Base* signifies that the algorithm *NewBase* extends algorithm *Base*.

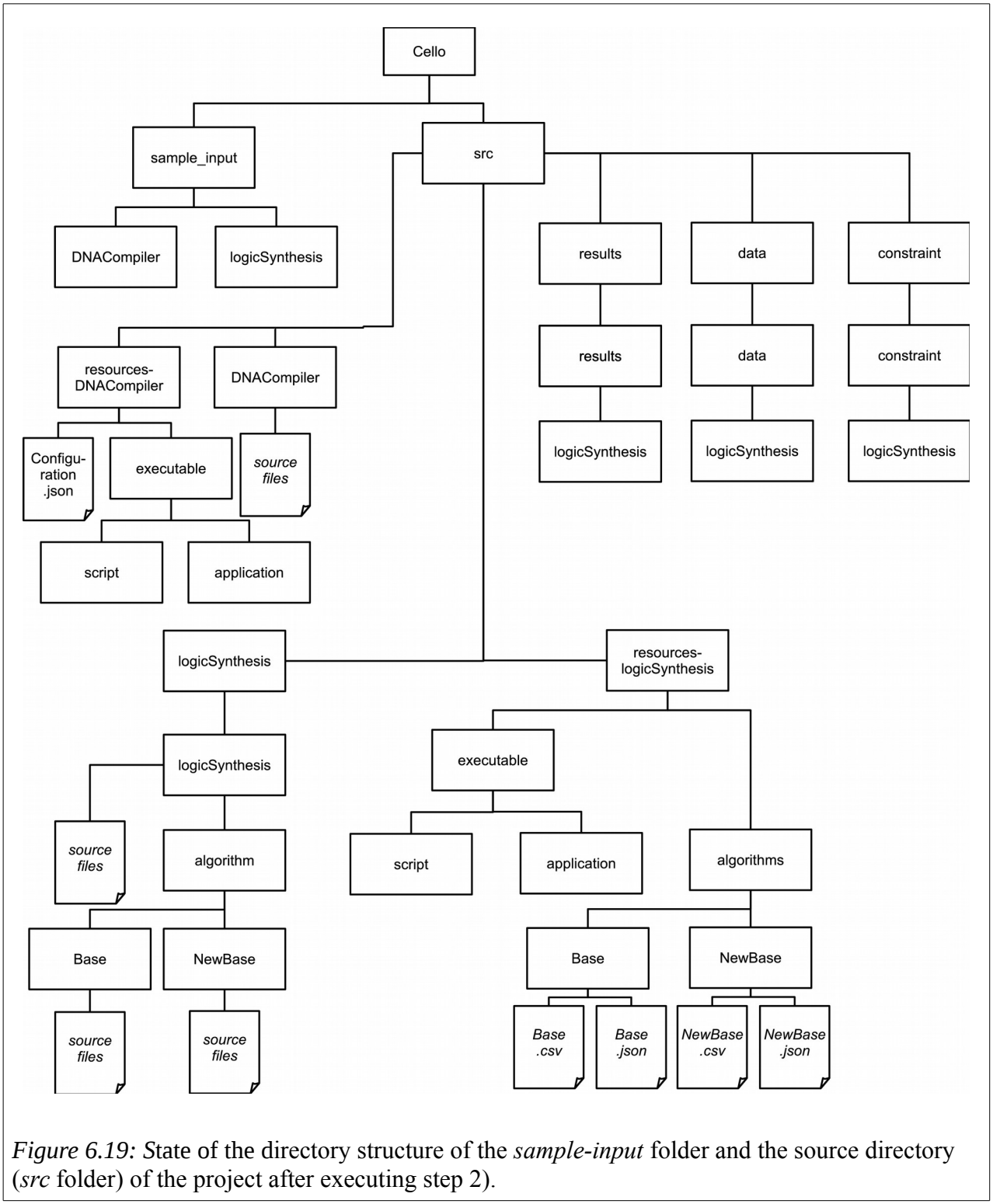

*Figure 6.19: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 2).

Figure 6.19 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 2).

3) ./add_stage_algo.py -i newAlgoParam.csv -p Cello/

AuthorName,*NewAuthor*,
ApplicationNames,*DNACompiler*,*other*
StagePrefix,*LS*,
StageName,*logicSynthesis*,
AlgorithmName,*NewParamBase*,
AlgorithmExtends,*Base*,
BooleanType,*TRUE*,boolean,
ByteType,*0*,byte,
CharType,*c*,char,
ShortType,*0*,short,
IntegerType,*0*,int,
LongType,*0*,long,
FloatType,*0.0*,float,
DoubleType,*0.0*,double,
StringType,*"str"*,string,
NewParam,"*newParam*",string,ADDED

Figure 6.20: INFILE file (newAlgoParam.csv) for step 3).

Figure 6.20 shows the content of the *INFILE* file (newAlgoParam.csv) used in step 3).

Description: The algorithm *NewParamBase* is added to the *logicSynthesis* stage in the *Cello* project. In addition, the algorithm *NewParamBase* has a new parameter **NewParam** of type string. For the *NewParamBase* algorithm, the generated algorithm-specific source files are found in *Cello/src/logicSynthesis/logicSynthesis/algorithm/NewParamBase*, its configuration file is found in *Cello/src/resources-logicSynthesis/algorithms/NewParamBase/NewParamBase.json*, and, its template file for extending the algorithm is found in *Cello/src/resources-logicSynthesis/algorithms/NewParamBase/NewParamBase.csv*. The *Cello/src/resources-logicSynthesis/algorithms/NewParamBase* directory is the directory for storing algorithm-specific files.

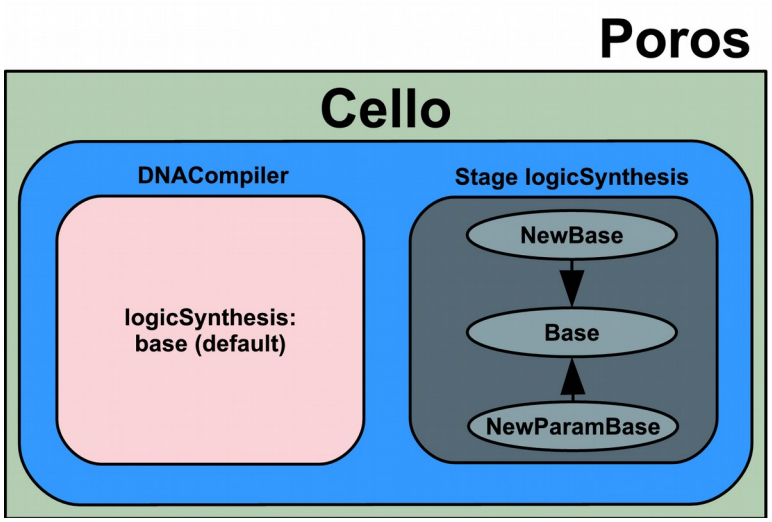

Figure 6.21: The state of the project after executing step 3).

Figure 6.21 shows the state of the project after executing step 3). The arrow between *NewParamBase* and *Base* signifies that the algorithm *NewParamBase* extends algorithm *Base*.

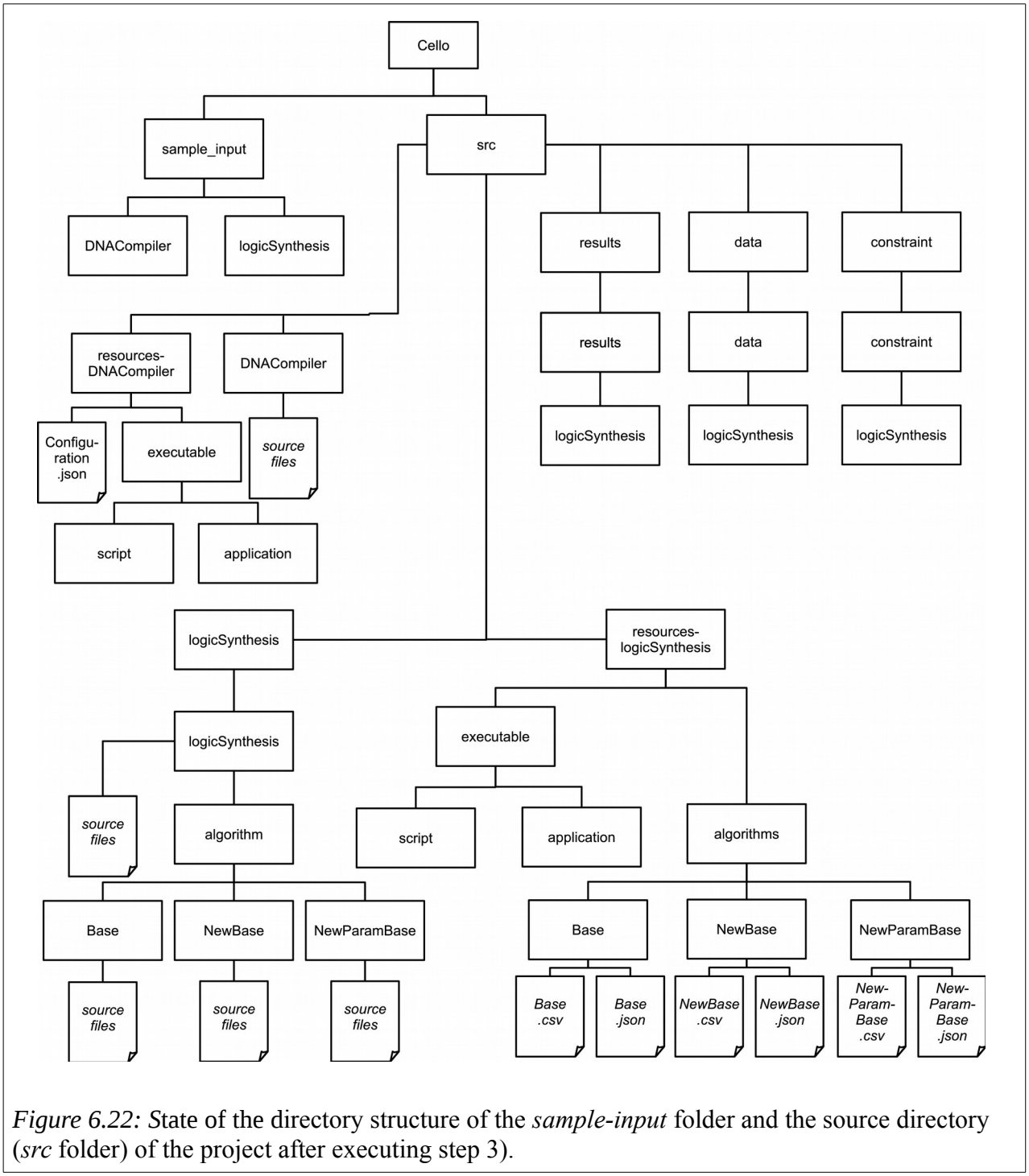

*Figure 6.22: S*tate of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 3).

Figure 6.22 shows the state of the directory structure of the *sample-input* folder and the source directory (*src* folder) of the project after executing step 3).

6.4 remove_stage_algo.py

remove_stage_algo.py is a python script that removes a stage and/or an algorithm from a project that uses the Poros framework.

When a stage is removed from a project, three directories are removed from the project: 1) the source directory containing the source files and infrastructure files for executing the stage, 2) the resource directory containing external files needed during the execution of the stage, and, 3) the directory for storing user-defined sample inputs for executing or testing the stage, this directory is found within the *sample-input* directory of the framework. These directories are the directories added to the project when a stage was added to the project. In addition to removing these directories, the tool removes: 1) the stages from the configuration file(s) of application(s), and, 2) the JAVA package in the *results*, *data* and *constraint* folder of the stage.

When an algorithm is removed from a stage of a project, two directories are removed from the project: 1) the directory containing the source files for the algorithm is added to the stage's source directory, and, 2) the directory containing algorithm-specific file(s), this directory is in the stage's resource directory. These directories are the directories added to the stage when an algorithm was added to the stage. In addition to removing the directory, the tool removes: 1) references to the algorithm-specific files in the stage's existing infrastructure files, 2) the algorithm's configuration file, and, 3) the template file for extending the algorithm.

6.4.1 Description

Help:

usage: remove_stage_algo.py [-h] -p PROJECTDIR -s STAGENAME [-a ALGONAME] -e

[Name [Name ...]]

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

-p PROJECTDIR, --projectDir PROJECTDIR

Project Directory

-s STAGENAME, --stageName STAGENAME

Stage Name

-a ALGONAME, --algoName ALGONAME

Algorithm Name

-e [Name [Name ...]], --appNames [Name [Name ...]]

Application Name(s)

Execution Signature:

./remove_stage_algo.py -p **PROJECTDIR** -s **STAGENAME** [-a **ALGONAME]** -e **[Name [Name ...]]**

-p PROJECTDIR (Required)

The PROJECTDIR describes the path to the root of the project that uses the Poros framework.

-s STAGENAME (Required)

STAGENAME corresponds to the name of the stage as it appeared in the **StageName** field when adding the stage to a project. Note: To remove the entirety of a stage, the -a option must **not** be specified.

-a ALGONAME (Optional)

The ALGONAME describes the name of the algorithm for removal from the project. If all algorithms within a stage or the default algorithm for a stage is removed, then no default algorithm exist for the stage. The user must manually define a default algorithm within the framework (Section 14.5.1).

-e [Name [Name ...]] (Optional)

The Name is the name of an application to remove the stage from its configuration. The stage is removed if the stage is marked for removal or if the default algorithm for a given stage is removed.

6.4.2 Example

For the following examples, the initial state of the project using the framework is shown in Figure 6.23. The framework contains a project with name *Cello* that contains an application name *DNACompiler*. The *Cello* project also contains two stages: *logicSynthesis* and *placement*. The *logicSynthesis* stage contains algorithms: *Base*, *NewBase* and *Other*. The placement stage contains algorithms *Base*, *NewBase* and *Placement*. In the *DNACompiler* application, the default algorithms for stages *logicSynthesis* and *placement* are *Base* and *Placement* respectively.

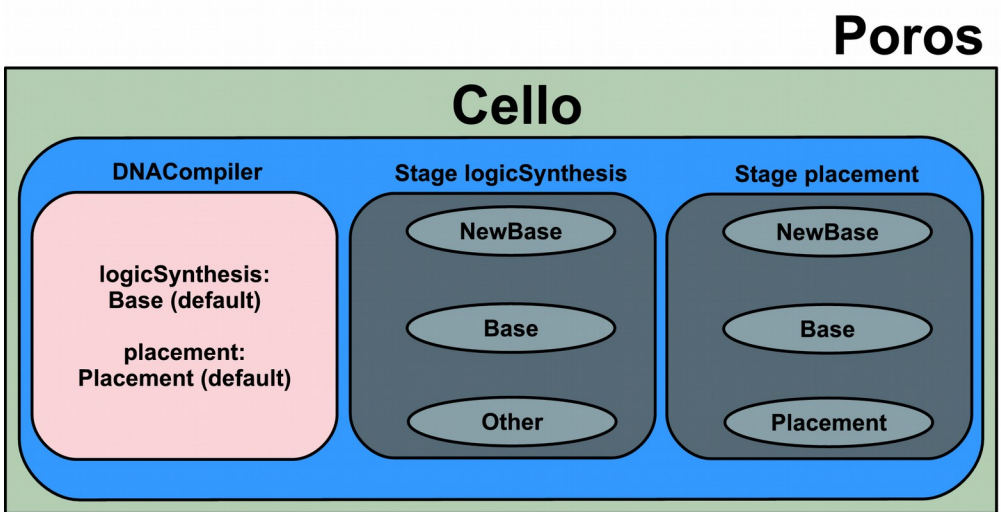

Figure 6.23: Initial state of the project using the framework

1) ./remove_stage_algo.py -p Cello/ -s logicSynthesis -a NewBase -e DNACompiler

Description: Remove algorithm *NewBase* from stage *logicSynthesis*. Note that there is no effect to the *DNACompiler* application, since the default algorithm for stage l*ogicSynthesis* is not algorithm *NewBase*. The following directories are removed from the project: *Cello/src/logicSynthesis/logicSynthesis/algorithm/NewBase* and *Cello/src/resources-logicSynthesis/algorithms/NewBase.*

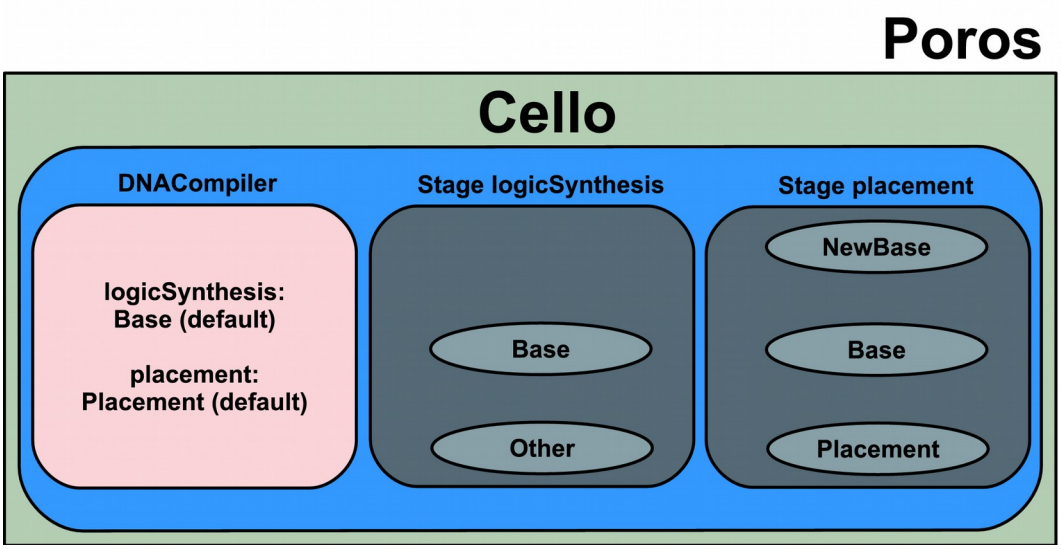

Figure 6.24: The state of the project after executing step 1).

Figure 6.24 shows the state of the framework after execution step 1).

2) ./remove_stage_algo.py -p Cello/ -s placement -a Base -e DNACompiler

Description: Remove algorithm *Base* from stage *placement*. Note that there is no effect to the *DNACompiler* application, since the default algorithm for stage *placement* is not algorithm *Base*. The following directories are removed from the project: *Cello/src/placement/placement/algorithm/Base* and *Cello/src/resources-placement/algorithms/Base.*

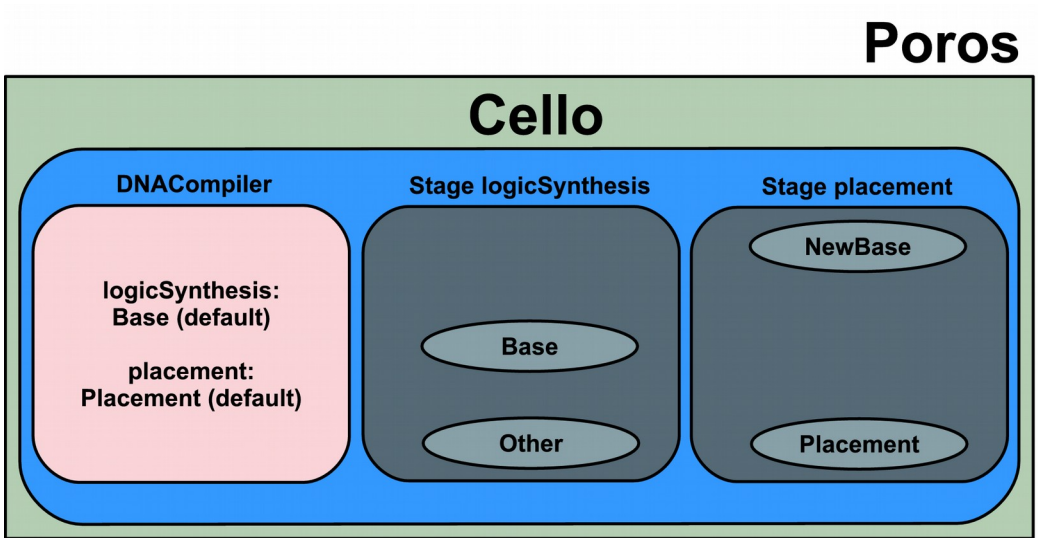

Figure 6.25: The state of the project after executing step 2).

Figure 6.25 shows the state of the project after executing step 2).

3) ./remove_stage_algo.py -p Cello/ -s logicSynthesis -e DNACompiler

Description: Remove the stage *logicSynthesis*. The framework updates the *DNACompiler* application's configuration by removing the stage from its configuration file (*Cello/src/resources-DNACompiler/Configuration.json*). The following directories are removed from the project: *Cello/src/logicSynthesis*, *Cello/src/resources-logicSynthesis, Cello/src/results/results/logicSynthesis*, *Cello/src/data/data/logicSynthesis*, and, *Cello/src/constraint/constraint/logicSynthesis*.

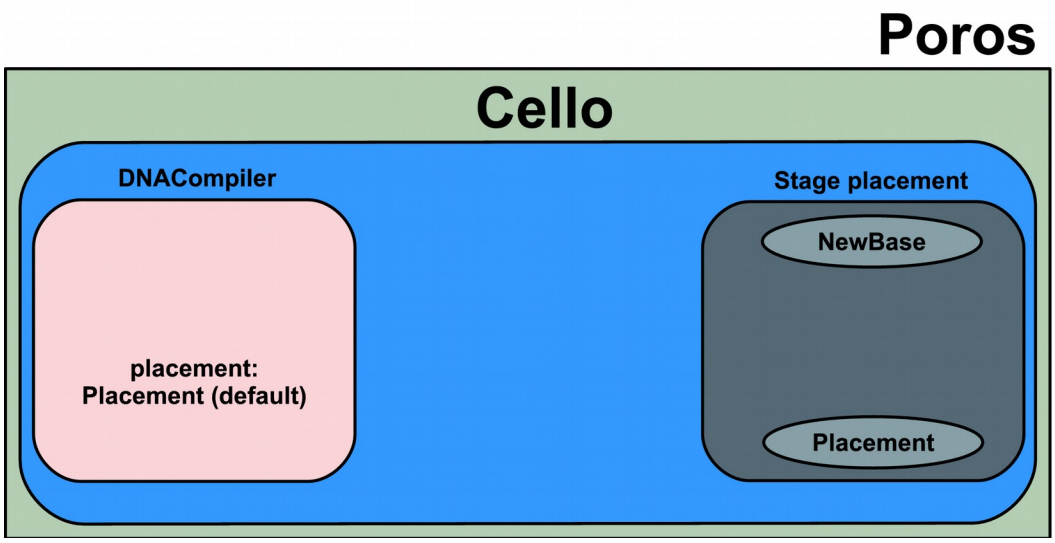

Figure 6.26: The state of the project after executing step 3).

Figure 6.26 shows the state of the project after executing step 3).

4) ./remove_stage_algo.py -p Cello/ -s placement -a Placement -e DNACompiler

Description: Remove algorithm *Placement* from stage *placement*. The framework updates the *DNACompiler* application's configuration by removing the stage from its configuration file *Cello/src/resources-DNACompiler/Configuration.json*). The following directories is removed from the project: *Cello/src/placement/placement/algorithm/Placement* and *Cello/src/resources-placement/algorithms/Placement.* However, the stage remains in the project.

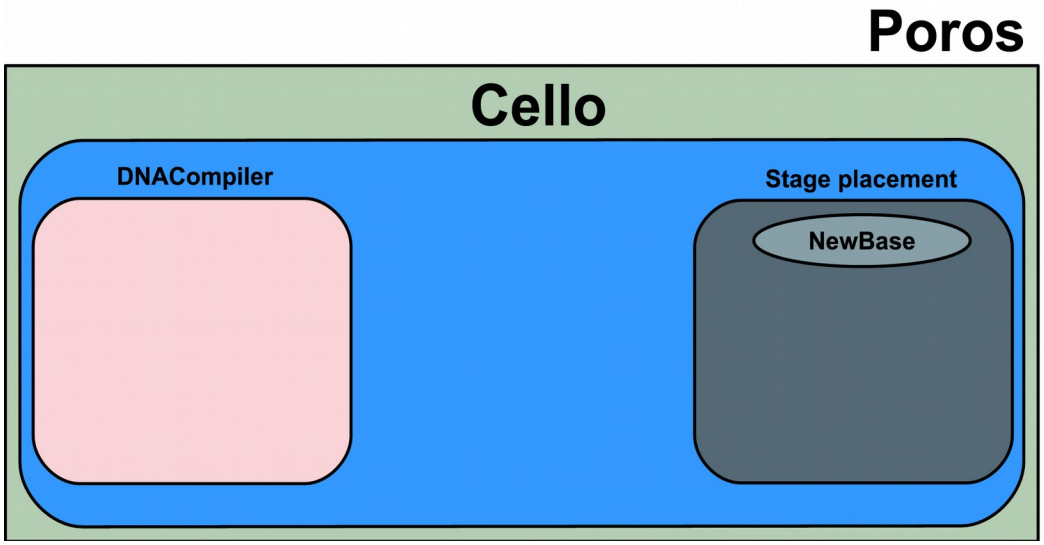

Figure 6.27: The state of the project after executing step 4).

Figure 6.27 shows the state of the project after executing step 4).

5) ./remove_stage_algo.py -p Cello/ -s placement -a NewBase -e DNACompiler

Description: Remove algorithm *NewBase* from stage *placement*. The following directories is removed from the project: *Cello/src/placement/placement/algorithm/NewBase* and *Cello/src/resources-placement/algorithms/NewBase.* Although the stage does not contain an algorithm, the stage remains in the project.

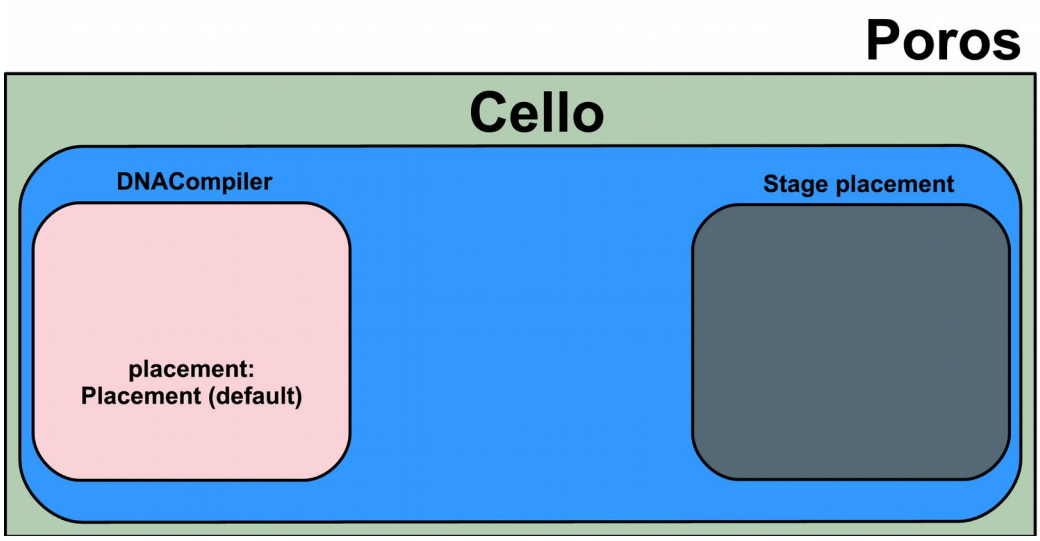

Figure 6.28: The state of the project after executing step 5).

Figure 6.28 shows the state of the project after executing step 5).

6.5 run.py

run.py is a python script that executes an application or a stage within the framework.

6.5.1 Description

Help:

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

run.py

-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. The arguments must be surround the arguments with double quotes (“”). Note that the *-cp* argument is reserved. To define a classpath, use the *-classpath* argument.

-e EXECUTABLE (Required)

The EXECUTABLE describes the name of the stage or application to execute.

-a EXEC_ARGS (Optional)

The EXEC_ARGS describes arguments of the executable. The arguments must be surround the arguments with double quotes (“”).

./remove_stage_algo.py -p Cello/ -s placement -a Placement -e DNACompiler

6.5.2 Example

For the following examples, the framework contains an application name *DNACompiler*.

1) ./run.py -e DNACompiler

   Description: Execute the DNACompiler executable.

2) ./run.py -e DNACompiler -a “-args arg.value”

   Description: Execute the DNACompiler executable with arguments *-args arg.value*.

3) ./run.py -j “-Xmx256m” -e DNACompiler -a “-args arg.value”

   Description: Execute the DNACompiler executable with arguments *-args arg.value*, and, java virtual machine arguments *-Xmx256m*.

7 Source Files

This chapter describes the source files of the framework. For more information, see the Application Programming Interface (API) documentation generated using Javadoc in the *PROJECT_DIRECTORY/Documentation/api/* directory, where *PROJECT_DIRECTORY* is the directory of the project. To display and navigate the documentation, open the *index.html* (*PROJECT_DIRECTORY/Documentation/api/common/index.html* or *PROJECT_DIRECTORY/Documentation/api/results/index.html*) in a web browser.

The source files are written in the Java programming language and are organized in Java packages that are found in the *src/common* and *src/results* folders. The following is a list of the packages and a description of their content:

- **common**: primitive data structures for the framework
- **common.algorithm**: base classes for algorithms created within a project
- **common.algorithm.data**: base classes for data used by algorithms created within a project
- **common.application**: base classes and utilities for executing an application
- **common.application.data**: base classes for data used within a netlist in a project
- **common.application.runtime.environment**: base classes for the runtime environment of an application
- **common.graph**: base classes for graph objects (data structure)
- **common.graph.algorithm**: algorithms for graph objects
- **common.graph.graph**: base classes for directed-edge graph data structure
- **common.graph.hypergraph**: base classes for hyper-edge graph data structure
- **common.JSON**: utilities for writing a JSON file
- **common.netlistConstraints.data**: base classes and utilities for netlist constraint data
- **common.options**: base classes and utilities for using the (optional) execution control file
- **common.profile**: base classes and utilities for parsing a JSON file/object
- **common.runtime**: base classes for executing a stage
- **common.runtime.environment**: primitive base classes for the runtime environment
- **common.stage**: base classes and utilities for executing a stage

- **common.stage.runtime.environment**: base classes for the runtime environment of a stage
- **common.target.data**: base classes and utilities for managing the target data
- **results.common**: utilities for the result package
- **results.netlist**: base classes for the netlist data structure
- **results.netlist.data**: base classes for the data of the netlist

8 Executable

There are two executable types generated by the framework: 1) a stage executable, and, 2) an application executable. The source for the executable is the *Main* class in the ***NAME_OF_EXECUTABLE****.runtime* Java package, where the ***NAME_OF_EXECUTABLE*** is the name of the executable. This package is located in the source folder corresponding to the executable, src/***NAME_OF_EXECUTABLE****,* where the ***NAME_OF_EXECUTABLE*** is the name of the executable.

8.1 Example

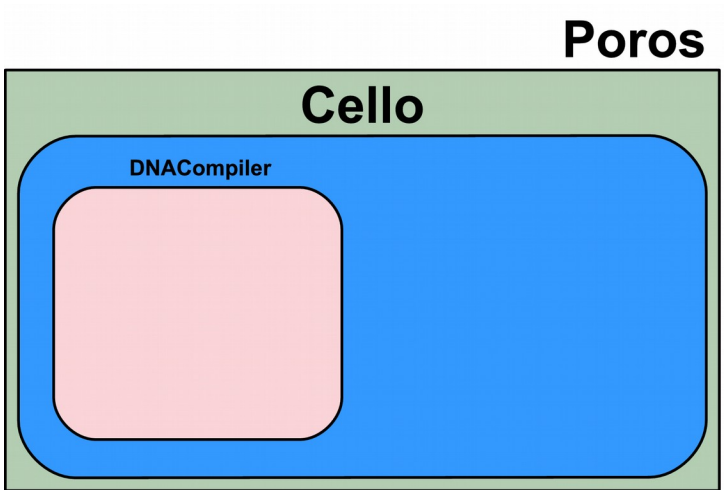

Figure 8.1: Initial state of a project using the framework

For the following example, the initial state of a project using the framework is shown in Figure 8.1. The framework contains a project with name *Cello* that contains an application name *DNACompiler*. Note the initial state of the project shown in Figure 8.1 is identical to the state of the project after executing step 1) in Section 6.2.2, illustrated in Figure 6.5. Thus, the source for the *DNACompiler* application executable is *Cello/src/DNACompiler/DNACompiler/runtime/Main.java*.

9 Execution Control File

The execution control file is used to modify the default execution of an application or a stage. The file overrides the default algorithm and/or its parameters. It is in comma-separated value (CSV) format.

9.1 File Format

Each line of the file is an entry that modifies the application's or stage's execution. Each entry is composed of two fields: 1) a name, and, 2) a value. The name is a reference to: 1) a stage instance, or, 2) an algorithm's parameter. The value is the new value associated with the name.

logicSynthesis,*NewBase*
logicSynthesis.**BooleanType**,*false*
logicSynthesis.**ByteType**,*1*,
logicSynthesis.**CharType**,*s*,
logicSynthesis.**ShortType**,*1*,
logicSynthesis.**IntegerType**,*1*,
logicSynthesis.**LongType**,*1*,
logicSynthesis.**FloatType**,1.*0*,
logicSynthesis.**DoubleType**,1.*0*,
logicSynthesis.**StringType**,*"str0"*

Figure 9.1: A sample execution control file

Figure 9.1 shows a sample execution control file. The first field of each entry is the name, and, the second field of each entry is the value. The names are in bold and the values are in italic.

To override the default algorithm of a stage, insert an entry into the execution control file where: 1) the name references the stage instance name, and, 2) the value contains the name of the *new* algorithm. In Figure 9.1, the first entry/line demonstrates overriding the default algorithm of the *logicSynthesis* instance with the *NewBase* algorithm.

To override the default parameter values of an algorithm, insert an entry into the execution control file where: 1) the name references the parameter by using the stage instance name and the parameter name as it appears in the *INFILE* (when the algorithm was created - Section 6.3) separated by a dot ("."), and, 2) the new value of the parameter. In Figure 9.1, the last nine entries/lines demonstrate overriding the default parameters of the *NewBase* algorithm of the logicSynthesis instance. For example, the new value of parameter **BooleanType** of algorithm *NewBase* is *false*.

9.2 Example

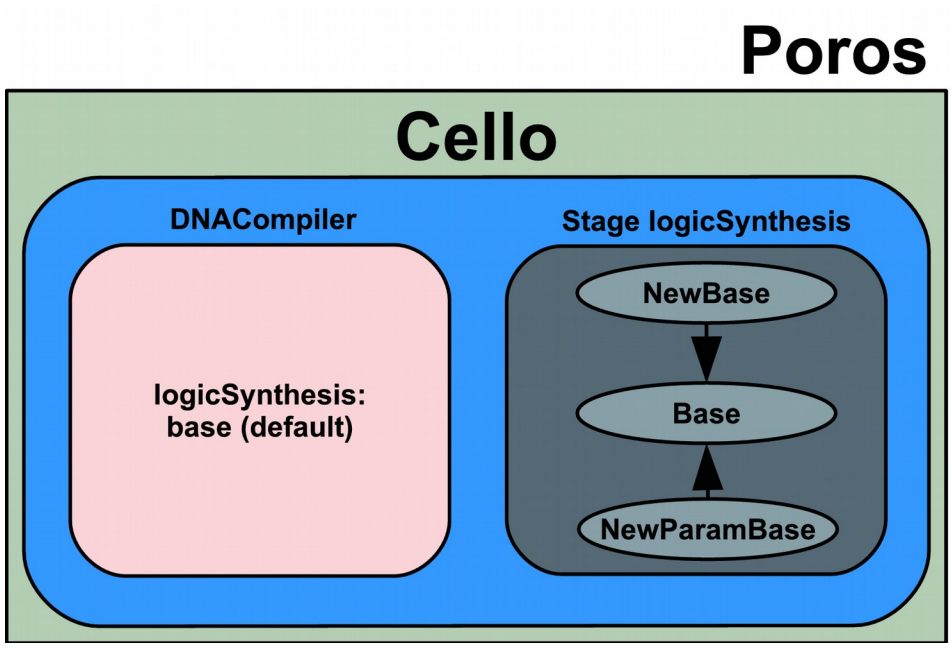

Figure 9.2: Initial state of the project using the framework.

For the following examples, the initial state of the project using the framework is shown in Figure 9.2. The framework contains a project with name *Cello* that contains an application name *DNACompiler*. The *Cello* project also contains the *logicSynthesis* stage with algorithms: *Base*, *NewBase* and *NewParamBase*. Algorithm *Base* contains a parameter named **doWork** of type boolean with default value *true*. The default algorithm for the *logicSynthesis* stage of the DNACompiler application is *Base*. Note that algorithms *NewBase* and *NewParamBase* extend algorithm *Base*, as shown with the arrows between *NewBase* and *Base*, and, *NewParamBase* and *Base*, as described in Section 6.3.

Override Algorithm

logicSynthesis,*NewBase*

Figure 9.3: Execution control file for overriding an algorithm

Figure 9.3 shows an example of an execution control file for overriding an algorithm. In the example, the execution control file overrides algorithm *Base* from stage *logicSynthesis*, enabling application *DNACompiler* to execute the *NewBase* algorithm.

Override Parameter

logicSynthesis.doWork,*false*

Figure 9.4: Execution control file for overriding a parameter

Figure 9.4 shows an example of an execution control file for overriding a parameter. In the example, the execution control file overrides parameter **doWork** from the algorithm of stage *logicSynthesis* with value *false*. Thus, the application *DNACompiler* executes the *Base* algorithm with parameter **doWork** set to *false*.

Override Algorithm and Parameter

logicSynthesis,*NewBase*
logicSynthesis.doWork,*false*

Figure 9.5: Execution control file for overriding an algorithm and a parameter

Figure 9.5 shows an example of an execution control file for overriding an algorithm and a parameter. In the example, the execution control file overrides: 1) algorithm *Base* from stage *logicSynthesis*, and, 2) parameter **doWork** from the algorithm of stage *logicSynthesis* with value *false.* Thus, the application *DNACompiler* executes the *NewBase* algorithm with parameter **doWork** set to *false*.

10 Command Line Arguments

The Poros framework enables the execution of an application and a stage. This Chapter describes: 1) the command line arguments present in the Poros framework, and, 2) the steps to manage, access and extend command line arguments from a programming perspective within the Poros framework.

10.1 Common command line arguments

There are seven common command line arguments for an executable (a stage or an application) in the Poros framework. Table 2 shows a summary of common command line arguments for an executable in the Poros framework. **Note**: For a stage executable, the file format for the parameter of *-inputNetlist* is constrained to the JSON format.

Argument	Description	Default value	Required	Number of Parameters
-help	print help message	N/A	No	N/A
-targetDataFile	path to target data file	N/A	Yes	1
-options	path to options file	N/A	No	1
-outputDir	path of output directory	Working directory	No	1
-pythonEnv	path to python environment	N/A	No	1
-inputNetlist	path to input netlist file	N/A	Yes	1
-outputNetlist	path to output netlist file	Filename is inputNetlist filename concatenated with "_outputNetlist.json", placed in *outputDir* directory.	No	1
-netlistConstraintFile	path to netlist constraint file	N/A	No	1
-logFilename[1]	log filename	Log.log	No	1

Table 2 Summary of common command line arguments for an executable the Poros framework

The common command line arguments for an executable are defined in the *ArgString* class of the *common.runtime.environment* Java package found in the *src/common* folder of the framework

1 Log file is placed into the *outputDir*

(*src/common/common/runtime/environment/ArgString.java*). The respective descriptions for each of these command line arguments are defined in the *ArgDescription* class of the *common.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/runtime/environment/ArgDescription.java*). These command line arguments are managed in the *RuntimeEnv* class of the *common.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/runtime/environment/RuntimeEnv.java*).

10.2 Stage command line arguments

There are two additional command line argument for a stage executable in the Poros framework: 1) a command line argument selecting the algorithm to execute within the stage, and, 2) a command line argument modifying the name of the stage instance. Table 3 shows a summary of the command line arguments for a stage executable in the Poros framework.

Argument	Description	Default value	Required	Number of Parameters
-algoName	algorithm name for Stage	N/A	Yes	1
-stageName	name of the Stage instance	N/A	No	1

Table 3 Summary of the command line arguments for a stage executable in the Poros framework

The common command line arguments for a stage are defined in the *StageArgString* class of the *common.stage.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/stage/runtime/environment/StageArgString.java*). The respective descriptions for each of these command line argument are defined in the *StageArgDescription* class of the *common.stage.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/stage/runtime/environment/StageArgDescription.java*). These command line arguments are managed in the *StageRuntimeEnv* class of the *common.stage.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/stage/runtime/environment/StageRuntimeEnv.java*).

The stage-specific command line arguments are defined in the ***STAGE_PREFIX****ArgString* class of the ***STAGE_NAME****.runtime.environment* Java package found in the *src/**STAGE_NAME*** folder of the framework (*src/**STAGE_NAME**/**STAGE_NAME**/runtime/environment/**STAGE_PREFIX**ArgString.java*), where ***STAGE_PREFIX*** and ***STAGE_NAME*** are the **StagePrefix** and **StageName** fields of an *INFILE* shown in Figure 6.11 in Section 6.3. The respective descriptions for each of these command line argument are defined in the ***STAGE_PREFIX****ArgDescription* class of the ***STAGE_NAME****.runtime.environment* Java package found in the *src/**STAGE_NAME*** folder of the framework

(*src/**STAGE_NAME**/**STAGE_NAME**/runtime/environment/**STAGE_PREFIX**ArgDescription.java*). These command line arguments are managed in the ***STAGE_PREFIX**RuntimeEnv* class of the ***STAGE_NAME**.runtime.environment* Java package found in the *src/**STAGE_NAME*** folder of the framework (*src/**STAGE_NAME**/**STAGE_NAME**/runtime/environment/**STAGE_PREFIX**RuntimeEnv.java*).

10.3 Application command line arguments

The common command line arguments for an application are defined in the *ApplicationArgString* class of the *common.application.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/application/runtime/environment/ApplicationArgString.java*). The respective descriptions for each of these command line argument are in defined the *ApplicationArgDescription* class of the *common.application.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/application/runtime/environment/Application/ArgDescription.java*). These command line arguments are managed in the *ApplicationRuntimeEnv* class of the *common.application.runtime.environment* Java package found in the *src/common* folder of the framework (*src/common/common/application/runtime/environment/ApplicationRuntimeEnv.java*).

The application-specific command line arguments are defined in the ***APPLICATION_NAME**ArgString* class of the ***APPLICATION_NAME**.runtime.environment* Java package found in the *src/**APPLICATION_NAME*** folder of the framework (*src/**APPLICATION_NAME**/**APPLICATION_NAME**/runtime/environment/**APPLICATION_NAME** ArgString.java*), where ***APPLICATION_NAME*** is the name of the application as referenced in the *APPNAME* value of the *-a* argument described in Section 6.2. The respective descriptions for each of these command line arguments are defined in the ***APPLICATION_NAME**ArgDescription* class of the ***APPLICATION_NAME**.runtime.environment* Java package found in the *src/**APPLICATION_NAME*** folder of the framework (*src/**APPLICATION_NAME**/**APPLICATION_NAME**/runtime/environment/**APPLICATION_NAME** ArgDescription.java*). These command line arguments are managed in the ***APPLICATION_NAME**RuntimeEnv* class of the ***APPLICATION_NAME**.runtime.environment* Java package found in the *src/**APPLICATION_NAME*** folder of the framework (*src/**APPLICATION_NAME**/**APPLICATION_NAME**/runtime/environment/**APPLICATION_NAME** RuntimeEnv.java*).

10.4 Managing, accessing and extending command line argument

This section provides the steps for managing, accessing and extending a command line argument. Managing a command line argument refers to modifying a property of an existing command line argument, where the property includes:

1) the command line argument,

2) the description of the command line argument, or,

3) an attribute of the command line argument.

The following source code modifications may be required:

1) overriding the command line argument,

2) overriding the description of the command line argument, or,

3) overriding the getter method for the command line argument.

The framework provides the *getOptionValue(str)* and the *getOptionValue(c)* methods to access the value of a command line argument, and, the *hasOption(str)* and the *hasOption(c)* methods to verify the presence of an argument during the execution of an executable. The *str* parameter is the long command line argument, and, the *c* parameter is the short command line argument. Additional details for these methods can be found in the Application Programming Interface (API) documentation (Chapter 7) of the *RuntimeEnv* class. **Note:** the command line arguments in the framework are long command line arguments.

To extend an executable with a new command line argument, the following source code modifications are required:

1) defining the new command line argument,

2) defining the description of the new command line argument,

3) adding a getter method for the new command line argument, and,

4) updating the *setOptions* method found in the *RuntimeEnv* class of the executable with the getter method.

10.5 Examples

This section shows examples for: 1) executing an executable, 2) managing command line arguments, 3) accessing command line arguments, and, 4) extending command line arguments. There are additional example for executing an executable in Section 6.5.

Executing an executable

1) EXECUTABLE -inputNetlist adder.v -targetDataFile ./Ecoli.UCF

   Description: Executing an application executable, *EXECUTABLE*, with: 1) input netlist *adder.v,* and, 2) the path to the target data file *./Ecoli.UCF*. This command has the minimal number of arguments for an application executable. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

2) EXECUTABLE -inputNetlist adder.v -targetDataFile ./Ecoli.UCF -options ./options.csv

Description: Executing an application executable, *EXECUTABLE*, with: 1) input netlist *adder.v*, 2) the path to the target data file *./Ecoli.UCF*, and, 3) the path to the options file *./options.csv*. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

3) EXECUTABLE -inputNetlist adder.v -targetDataFile ./Ecoli.UCF -outputNetlist ./adder_output.json

   Description: Executing an application executable, *EXECUTABLE*, with: 1) input netlist *adder.v,* and, 2) the path to the target data file *./Ecoli.UCF*. The output netlist is located at *WORKING_DIRECTORY /./adder_output.json*.

4) EXECUTABLE -inputNetlist adder.v -targetDataFile ./Ecoli.UCF -outputDir ../../results -outputNetlist ./adder_output.json

   Description: Executing an application executable, *EXECUTABLE*, with: 1) input netlist *adder.v,* 2) the path to the target data file *./Ecoli.UCF*, and, 3) the path to the output directory *../../results*. The output netlist is located at *../../results/./adder_output.json*.

5) EXECUTABLE -inputNetlist adder.v -targetDataFile ./Ecoli.UCF -pythonEnv /usr/bin/python2.7

   Description: Executing an application executable, *EXECUTABLE*, with input: 1) netlist *adder.v*, 2) the path to the target data file *./Ecoli.UCF*, and, 3) the path to the python environment */usr/bin/python2.7*. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

6) EXECUTABLE -inputNetlist adder.json -targetDataFile ./Ecoli.UCF -algoName Base

   Description: Executing a stage executable, *EXECUTABLE*, with input netlist *adder.v,* the path to the target data file *./Ecoli.UCF*, and, the algorithm name to execute *Base*. This command has the minimal number of arguments for a stage executable. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

7) EXECUTABLE -inputNetlist adder.json -targetDataFile ./Ecoli.UCF -algoName Base -pythonEnv /usr/bin/python2.7

   Description: Executing a stage executable, *EXECUTABLE*, with input: 1) netlist *adder.v*, 2) the path to the target data file *./Ecoli.UCF*, 3) the algorithm name to execute *Base*, and, 4) the path to the python environment */usr/bin/python2.7*. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

8) EXECUTABLE -inputNetlist adder.json -targetDataFile ./Ecoli.UCF -algoName Base -logFilename Base.log

   Description: Executing a stage executable, *EXECUTABLE*, with input: 1) netlist *adder.v*, 2) the path to the target data file *./Ecoli.UCF*, 3) the algorithm name to execute *Base*, and, 4) the log

filename defined as *Base.log*. The output netlist is located at *WORKING_DIRECTORY/adder_outputNetlist.json*.

Managing, accessing and extending command line argument

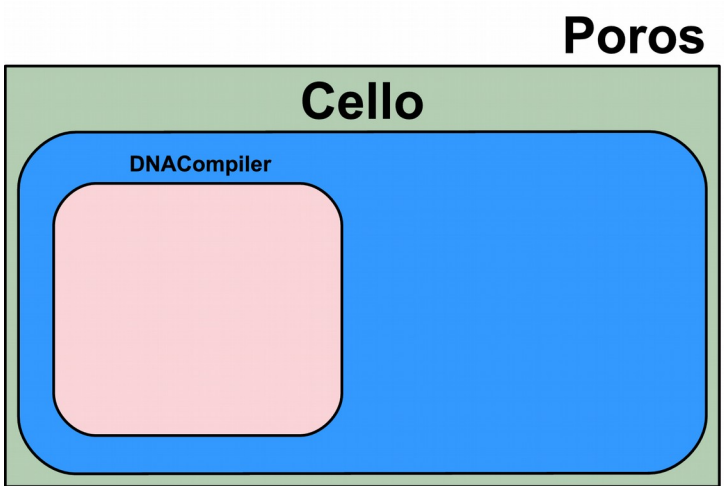

Figure 10.1: Initial state of the project using the framework

For the following examples (*Managing command line arguments*, *Accessing command line arguments*, and, *Extending command line arguments*), the initial state of the project using the framework is shown in Figure 10.1. The framework contains a project with name *Cello* that contains an application name *DNACompiler*. Note the initial state of the project shown in Figure 10.1 is identical to the state of the project after executing step 1) in Section 6.2.2, illustrated in Figure 6.5.

Managing command line arguments

The following example describes the steps to modify the string reference and the description of a command line argument. In the following example, the modifications are applied to an application, *DNACompiler*. For the *DNACompiler* application, the *-inputNetlist* string reference is modified to *-verilogFile*, and, the description for the *-inputNetlist* string reference is modified to "*path to input netlist file (in verilog format)*".

1) Override the contents of the variable for the *-inputNetlist* string reference.

```
public class DNACompilerArgString extends ApplicationArgString{

        final static public String INPUTNETLIST = "verilogFile";

}
```

Figure 10.2: Modification to the *DNACompilerArgString* class.

Figure 10.2 shows the modification to the *DNACompilerArgString* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerArgString.java*. The modification is shown in bold.

2) Override the contents of the variable describing the *-inputNetlist* string reference

```
public class DNACompilerArgDescription extends ApplicationArgDescription{

        final static public String INPUTNETLIST_DESCRIPTION = "path to
input netlist file (in verilog format)";

}
```

Figure 10.3: Modification to the *DNACompilerArgDescription* class.

Figure 10.3 shows the modification to the *DNACompilerArgDescription* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerArgDescription.java*. The modification is shown in bold.

3) Override the getter method of the *-inputNetlist* command line argument

```
public class DNACompilerRuntimeEnv extends ApplicationRuntimeEnv{

        @Override
        protected Option getInputNetlistOption(){
                Option rtn = new
Option( DNACompilerArgString.INPUTNETLIST, true,
DNACompilerArgDescription.INPUTNETLIST_DESCRIPTION);
                this.makeRequired(rtn);
                return rtn;
        }
}
```

Figure 10.4: Modification to the *DNACompilerRuntimeEnv* class.

Figure 10.4 shows the modification to the *DNACompilerRuntimeEnv* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerRuntimeEnv.java*. The modification is shown in bold.

Accessing command line arguments

This section shows an example for accessing the *-inputNetlist* command line argument in the *DNACompiler* application.

```
import java.io.File;
import DNACompiler.runtime.environment.DNACompilerArgString;
import DNACompiler.runtime.environment.DNACompilerRuntimeEnv;
...
public class Main {
        public static void main(String[] args) {
                // RuntimeEnv
                DNACompilerRuntimeEnv runEnv = new
DNACompilerRuntimeEnv(args);
                runEnv.setName("DNACompiler");
                if (!runEnv.isValid()) {
                        throw new RuntimeException("DNACompilerRuntimeEnv is
invalid!");
                }
                ...
                /*
                 * Get InputFile from user
                 */
                // InputFile
                String inputFilePath =
runEnv.getOptionValue(DNACompilerArgString.INPUTNETLIST);
                File inputFile = new File(inputFilePath);
                if (!(inputFile.exists() && !inputFile.isDirectory())) {
                        throw new RuntimeException("Input file does not exist!");
                }
                ...
        }
}
```

Figure 10.5: Example of accessing the *-inputNetlist* command line argument in the *DNACompiler* application

Figure 10.5 shows an example of accessing the *-inputNetlist* command line argument in the *DNACompiler* application. The source code is located at *Cello/src/DNACompiler/DNACompiler/runtime/Main.java*. In the *main* method of the *Main* class, the source code for accessing the *-inputNetlist* command line argument is shown in bold. The value associated with the *-inputNetlist* command line argument is stored in the *inputFilePath* variable of the *main* method. The code in the *main* method before the bold prepares an instance of the *DNACompilerRuntimeEnv* class to access the value of the *-inputNetlist* command line argument.

Extending command line arguments

The following example describes the steps for extending an executable with a new command line. In the following example, the *DNACompiler* application is extended with a command line argument, *-debug*. This command line argument is not required and is associated with a single value.

1) Defining the new command line argument

```
public class DNACompilerArgString extends ApplicationArgString{

        final static public String DEBUG = "debug";

}
```

Figure 10.6: Definition of the *-debug* command line argument in the *DNACompilerArgString* class.

Figure 10.6 shows the definition of the *-debug* command line argument in the *DNACompilerArgString* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerArgString.java*. The definition is shown in bold.

2) Defining the description of the new command line argument

```
public class DNACompilerArgDescription extends ApplicationArgDescription{

        final static public String DEBUG_DESCRIPTION = "debug level";

}
```

Figure 10.7: Definition of the description for *-debug* command line argument in the *DNACompilerArgDescription* class.

Figure 10.7 shows the definition of the description for *-debug* command line argument in the *DNACompilerArgDescription* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerArgDescription.java*. The definition is shown in bold.

3) Adding a getter method for the new command line argument

```
public class DNACompilerRuntimeEnv extends ApplicationRuntimeEnv{
        ...
        protected Option getDebugOption(){
                Option rtn = new Option( DNACompilerArgString.DEBUG,
true, DNACompilerArgDescription.DEBUG _DESCRIPTION);
                return rtn;
        }
        ...
}
```

Figure 10.8: Modification to the *DNACompilerRuntimeEnv* class for adding the getter method of the *-debug* command line argument.

Figure 10.8 shows the modification to the *DNACompilerRuntimeEnv* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerRuntimeEnv.java,* for adding the getter method of the *-debug* command line argument. The modification is shown in bold.

4) Updating the *setOptions* method found in the *DNACompilerRuntimeEnv* class with the getter method

```
import org.apache.commons.cli.Options;
…
public class DNACompilerRuntimeEnv extends ApplicationRuntimeEnv{
        …
        @Override
        protected void setOptions() {
                super.setOptions();
                // uncomment the line below to add options
                Options options = this.getOptions();
                options.addOption(this.getDebugOption());
        }
}
```

Figure 10.9: Modification to update the *setOptions* method found in the *DNACompilerRuntimeEnv* class with the getter method of the *-debug* command line argument.

Figure 10.9 shows the modification to update the *setOptions* method found in the *DNACompilerRuntimeEnv* class, located at *Cello/src/DNACompiler/DNACompiler/runtime/environment/DNACompilerRuntimeEnv.java,* with the getter method of the *-debug* command line argument. The modification is shown in bold.

11 User Design

The user design describes the input circuit to the framework. For an application, the file format of the user design is not restricted. However, for a stage, the file format of the user design is restricted to the JSON format.

From a programming perspective, the user design must be translated to an instance of the *Netlist* class. A *Netlist* instance is composed of instances of the *NetlistNode* and the *NetlistEdge* classes. An instance of the *NetlistNode* class contains a list of instances of the *NetlistEdge* class, and, an instance of the *NetlistEdge* class contains a list of instances of the *NetlistNode* class. These classes are found in the *results.netlist* package in the *src/results* folder of the framework.

The *Netlist, NetlistNode* and *NetlistEdge* classes contain data to store the results of the stages. The data are stored in the *ResultNetlistData*, *ResultNetlistNodeData* and *ResultNetlistEdgeData* for the *Netlist, NetlistNode* and *NetlistEdge* classes respectively. These classes require modification when adding a new stage to the project. These modifications includes: 1) adding an attribute that is a type equivalent to a primitive type from the Java programming language (e.g. boolean, byte, char, short, int, long, float, double) or a String, 2) adding the respective methods for accessing this attribute, also known as *getter* and *setter* methods, 3) defining a default value for the attribute, 4) extending the class to extract the attribute from an instance of the *JSONObject* class, and, 5) extending the class to write for the attribute in JSON format.

11.1 Examples

Netlist Representation In Software

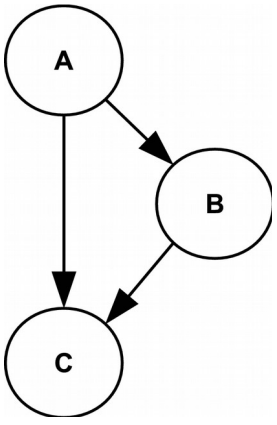

Figure 11.1: Example of a netlist representation in software

Figure 11.1 shows an example of a netlist representation in software. The Figure consist of an instance of the *Netlist* class with: 1) three instance of the *NetlistNode* class: *A*, *B*, and, *C*, and, 2) three instances of the NetlistEdge class:

a) an instance between the *A* and *B* instances of the *NetlistNode* class, where instance *A* is the source node and instance *B* is the destination node,

b) an instance between the *A* and *C* instances of the *NetlistNode* class, where instance *A* is the source node and instance *C* is the destination node, and,

c) an instance between the *B* and *C* instances of the *NetlistNode* class, where instance *B* is the source node and instance *C* is the destination node.

Adding an attribute to the NetlistNode class

The following example demonstrate the steps for adding an attribute to the data instance, *ResultNetlistNodeData*, of a *NetlistNode* class. These steps can be applied to the *Netlist* and *NetlistEdge* classes for adding an attribute within these classes.

For the following example, the following modification is associated with adding the *logicSynthesis* stage to a project. The example is a precursor to executing step 1) in Section 6.3.2. The *logicSynthesis* stage is responsible for assigning the node type to the instance of the *NetlistNode* class. Hence, an attribute with name *nodeType* is added to the class.

1) Adding the *nodeType* attribute to the *ResultNetlistNodeData* class

```
public class ResultNetlistNodeData ...{
        ...
        private String nodeType;
        ...
}
```

Figure 11.2: Modification to the *ResultNetlistNodeData* class for adding the *nodeType* attribute

Figure 11.2 shows the modification to the *ResultNetlistNodeData* class for adding the *nodeType* attribute. The modification is shown in bold.

2) Adding the methods for accessing the *nodeType* attribute to the *NetlistNode* class

```
public class ResultNetlistNodeData ...{
        ...
        public void setNodeType(String nodeType) {
                this.nodeType = nodeType;
        }

        public String getNodeType() {
                return this.nodeType;
        }
        ...
}
```

Figure 11.3: Modification to the *ResultNetlistNodeData* class for adding the methods for accessing the *nodeType* attribute

Figure 11.3 shows the modification to the *ResultNetlistNodeData* class for adding the methods for accessing the *nodeType* attribute. The modification is shown in bold.

3) Defining a default value for the *nodeType* attribute

```
public class ResultNetlistNodeData ...{
        ...
        private void setDefault() {
                this.setNodeType("");
        }
        ...
}
```

Figure 11.4: Modification to the *ResultNetlistNodeData* class for defining a default value of the *nodeType* attribute

Figure 11.4 shows the modification to the *ResultNetlistNodeData* class for defining a default value of the *nodeType* attribute. The modification is shown in bold.

4) Extending the class to extract the *nodeType* attribute from an instance of the *JSONObject* class

```
public class ResultNetlistNodeData ...{
        ...
        private void parseNodeType(final JSONObject JObj){
                String nodeType = ProfileUtils.getString(JObj,
"nodeType");
                if (nodeType != null) {
                        this.setNodeType(nodeType);
                }
        }

        private void parse(final JSONObject JObj){
                this.parseName(JObj);
                this.parseNodeType (JObj);
        }
        ...
}
```

Figure 11.5: Modification to the *ResultNetlistNodeData* class for extending the class to extract the *nodeType* attribute from a *JSONObject*

Figure 11.5 shows the modification to the *ResultNetlistNodeData* class for extending the class to extract the *nodeType* attribute from a *JSONObject*. The modification is shown in bold.

5) Extending the class to write for the *nodeType* in JSON format

```
public class ResultNetlistNodeData ...{
        ...
        public void writeJSON(int indent, Writer os) throws
IOException{
                String str = "";
                str += JSONUtils.getEntryToString("nodeType",
this.getNodeType());
                str = JSONUtils.addIndent(indent, str);
                os.write(str);
        }
        ...
}
```

Figure 11.6: Modification to the *ResultNetlistNodeData* class for extending the class to write for the the *nodeType* in JSON format

Figure 11.6 shows the modification to the *ResultNetlistNodeData* class for extending the class to write for the the *nodeType* in JSON format. The modification is shown in bold.

5) Extending the copy constructor with the *nodeType*

```
public class ResultNetlistNodeData ...{
        ...
        public void ResultNetlistNodeData(ResultNetlistNodeData
other){
                this.setNodeType(other.getNodeType());
        }
        ...
}
```

Figure 11.7: Modification to the *ResultNetlistNodeData* class for extending the copy constructor with the *nodeType*

Figure 11.7 shows the modification to the *ResultNetlistNodeData* class for extending the class for extending the copy constructor with the *nodeType*. The modification is shown in bold.

12 User Design Constraints

The user design constraints is an input to the framework. The user design constraints describes constraints on the user design. These constraints are design-dependent.

The user constraint data file contains design- dependent data used by the application to derive the implementation for a user design and its constraints. The file is in JavaScript Object Notation[1] (JSON) format. For a description of the JSON format, the user is referred to https://www.json.org/.

The software representation of the user constraint data file is the *NetlistConstraint* class. The software representation of a constraint for a stage should be placed in the *src/constraint/constraint/**STAGE_NAME*** folder enabling other stages to reference them, where **STAGE_NAME** refers to the name of the stage.

12.1 File Format

The file format is identical to that of the target data (Section 13.1).

12.2 Example

Example from the target data can be applied for the user constraint data file (Section 13.2).

1 https://www.json.org/

13 Target Data

The target data file contains target-specific data used by the application to derive the implementation for a user design and its constraints. The file is in JavaScript Object Notation[1] (JSON) format. For a description of the JSON format, the user is referred to https://www.json.org/.

The software representation of the target data file is the *TargetData* class. The software representation of data for a stage should be placed in the *src/dat/data/**STAGE_NAME*** folder enabling other stages to reference them, where **STAGE_NAME** refers to the name of the stage.

13.1 File Format

The data in the file is represented by a JSON array. Each element in the array is a data instance. The various data types are referenced using the *collection* attribute of an element.

```
[
        {
                "collection": "gates",
                "number": 2
        },
        {
                "collection": "gate-type",
                "name": "2AND1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        },
        {
                "collection": "gate-type",
                "name": "2OR1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        }
]
```

Figure 13.1: A sample target data file

Figure 13.1 shows a sample target data file. In the sample file, there are three entries: one entry of collection *gates* and two entries of collection *gate-type*.

13.2 Example

For the following examples, the initial state of the target data file is shown in Figure 13.1.

Insert new entry with an existing collection (data type)

1 https://www.json.org/

To insert a new entry of an existing collection (data type), insert a new element in the array in the target data file with the value of the *collection* attribute equivalent to an existing collection.

```
[
        {
                "collection": "gates",
                "number": 2
        },
        {
                "collection": "gate-type",
                "name": "2AND1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        },
        {
                "collection": "gate-type",
                "name": "2OR1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        },
        {
                "collection": "gate-type",
                "name": "1NOT1",
                "number_of_inputs": 1,
                "number_of_outputs": 1
        }
]
```

Figure 13.2: Inserting a new entry with an existing collection (data type) in the target data

Figure 13.2 shows an example of inserting a new entry with an existing collection (data type) in the target data of Figure 13.1. The entry is of collection (data type) *gate-type* the entry name is *1NOT1* The changes to the file are shown in bold.

Insert new collection (data type)

To insert a new entry with a new collection (data type), insert a new element in the array in the target data file with a unique value for the collection attribute. A unique value signifies that no other entry within the array contains the given value for the *collection* attribute.

```
[
        {
                "collection": "gates",
                "number": 2
        },
        {
                "collection": "gate-type",
                "name": "2AND1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        },
        {
                "collection": "gate-type",
                "name": "2OR1",
                "number_of_inputs": 2,
                "number_of_outputs": 1
        },
        {
                "collection": "input-type",
                "name": "ExternalPin"
        }
]
```

Figure 13.3: Inserting a new entry with a new collection (data type) in the target data

Figure 13.2 shows an example of inserting a new entry with a new collection (data type) in the target data file of Figure 13.1. The new collection (data type) is *input-type*. A single entry is present in the target data, it has the name *ExternalPin*. The changes to the file are shown in bold.

Accessing the entries in the target data file using the Poros framework

In the Poros framework, the target data file is represented by a *TargetData* object instance. To access the entries in the target data file, the framework provides two methods: 1) *getNumJSONObject(type)*, and, 2) *getJSONObjectAtIdx(type, index)*. The method *getNumJSONObject* requires the *collection* (data type) as a parameter and returns the number of elements in the array with the given type. The method *getJSONObjectAtIdx* requires the *collection* (data type) and index as parameters, and, returns the *JSONObject* representing the element of the given *collection* (data type) at the given *index*.

```
import common.target.data.TargetData;
import org.json.simple.JSONObject;
...
public class ... {

        public void methodAccessingTargetData(TargetData td){
                for (int i = 0; i < td.getNumJSONObject("gate-type"); i++){
                        JSONObject jObj = td.getJSONObjectAtIdx("gate-type", i)
                        <PROCESS_JSONOBJECT_jObj>
                }
        }

}
```

Figure 13.4: Example of accessing the *gate-type* entries in the target data file using the Poros framework

Figure 13.4 shows an example of accessing the *gate-type* entries in the target data file using the Poros framework. Upon retrieving a *JSONObject*, the *JSONObject* instance can be used for data processing or computation. However, it is recommended to translate the *JSONObject* to a Java class Object for processing.

14 Programming in Poros

This Chapter presents the details of the Poros framework from a programming perspective. For details on programming with the Java programming language, the reader is referred to the following references:

1) http://www.oracle.com/technetwork/topics/newtojava/
2) Kathy Sierra and Bert Bates. *Head First Java*. 2nd Edition. O'Reilly Media. 2005.
3) Bruce Eckel. *Thinking in Java*. 4th Edition. Prentice Hall. 2006.
4) Herbert Schildt. *Java: The Complete Reference*. 10th Edition. McGraw-Hill Education. 2017.
5) Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. 1st Edition. Addison-Wesley Professional. 2005.

For details on using the Eclipse development environment, the reader is referred to the following references:

1) https://www.tutorialspoint.com/eclipse/
2) https://www.udemy.com/eclipse-java-tutorial-for-beginners/

For details on the JavaScript Object Notation (JSON) format, the reader is referred to the following references:

1) https://www.json.org/.

14.1 Application

For an application, the framework generates infrastructure files for: 1) Managing, accessing and extending command line arguments (Section 10.3 and 10.4), and, 2) the executable (Chapter 8).

The purpose of an application is to define the default execution flow. When the application is executed as an executable (Chapter 8), the default execution flow is defined in the *main* method of the *Main* class. From a programming perspective, the default execution flow is defined by inserting stages into the *main* method. The framework provides a tool for adding a stage to a project (Section 6.3). When the tool is invoked, the tool displays instructions with the source code on inserting a stage into an application. The source code should be placed below the comment *Add Stages below* and above the comment *Add Stages above* in the *main* method of the *Main* class.

The default execution flow in the generated infrastructure file for an application consist of:

1) initializing an instance of the *RuntimeEnv* class for the application,
2) initializing an instance of the *Netlist* class,
3) initializing an instance of the *TargetData* class,
4) initializing an instance of the *NetlistConstraint* class,
5) retrieving the user design from the command line arguments,
6) setting the user design attribute to the netlist instance,

7) executing the stage(s), and,

8) writing the netlist to an output file.

Note that 6) refers to the location for inserting the stage into an application.

```
...
public class Main {
        public static void main(String[] args) {
                ...
                /*
                 * Add Stages below
                 */
                <INSERT_CODE_FOR_STAGES>
                /*
                 * Add Stages above
                 */
                ...
        }
}
```

Figure 14.1: Region to insert the code for a stage. The region is highlighted by the ***<INSERT_CODE_FOR_STAGES>*** tag.

Figure 14.1 shows the region to insert the code for a stage. The region is highlighted by the ***<INSERT_CODE_FOR_STAGES>*** tag.

14.1.1 Example

For the following example, the tool described in Section 6.3 is invoked with the arguments of step 1) from Section 6.3.2.

```
+------------------------------------------------------------------------
|       IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT!
|       Instructions:
|                       To use the stage in the main execution flow of
|                       an application ${APPLICATION_NAME}, where
|                       ${APPLICATION_NAME} is one of:
|                       DNACompiler
|                       other
|
|                       copy the following snippet of JAVA code and paste it
|                       into the 'main' function of the applications JAVA file
|                       (${PROJECT_DIRECTORY}/src/${APPLICATION_NAME}/
|                       ${APPLICATION_NAME}/runtime/Main.java)
+------------------------------------------------------------------------
                // logicSynthesis
                currentStage = appCfg.getStageByName("logicSynthesis");
                LSRuntimeObject LS = new LSRuntimeObject(currentStage, td, netlistConstraint,
netlist, runEnv);
                LS.execute();
```

Figure 14.2: Snippet of the instructions generated by the tool described in Section 6.3 after executing step 1) from Section 6.3.2

Figure 14.2 shows a snippet of the instructions generated by the tool described in Section 6.3 after executing step 1) from Section 6.3.2. The instructions state that the snippet of code shown in bold should be placed in the *main* method of the *Main* class for the applications: *DNACompiler* and *other*. This source code should be placed below the comment *Add Stages below* and above the comment *Add Stages above*, as highlighted in Figure 14.1. Note the *"logicSynthesis"* parameter in the *getStageByName* method for the *appCfg* instance refers to the stage name as described in the application configuration file (Section 14.5). The code initializes an instance of the stage, then, executes the instance.

```
...
public class Main {
        public static void main(String[] args) {
                ...
                /*
                 * Add Stages below
                 */
                // logicSynthesis
                currentStage = appCfg.getStageByName("logicSynthesis");
                LSRuntimeObject LS = new LSRuntimeObject(currentStage, td, netlistConstraint,
netlist, runEnv);
                LS.execute();
                /*
                 * Add Stages above
                 */
                ...
        }
}
```

Figure 14.3: Modifications to the *main* method of the *Main* class for an application when inserting the source code from Figure 14.2

Figure 14.3 shows the modification to the *main* method of the *Main* class for an application when inserting the source code from Figure 14.2 in the region highlighted by Figure 14.1. The modification is shown in bold.

14.2 Stage

For a stage, the framework generates infrastructure files for: 1) Managing, accessing and extending command line arguments (Section 10.2 and 10.4), and, 2) the executable (Chapter 8).

The purpose of a stage is to: 1) compute a result for an attribute in the netlist that is used in future stages, and, 2) select the algorithm for computing the result. Details for modifying a netlist, the software representation of a user design, is discussed in Chapter 11.

When the stage is executed as an executable (Chapter 8), the default execution flow of the stage is defined in the *main* method of the *Main* class. The framework provides the infrastructure for a default execution flow for a stage. The default execution should be modified on rare occasions such as extending the stage with a new command line argument. From a programming perspective, details for extending the stage with a new command line argument are found in Section 10.4 with an example discussed in Section 10.5.

The default execution flow in the generated infrastructure file for a stage consist of:

1) initializing an instance of the *RuntimeEnv* class for the stage,
2) retrieving the user design from the command line arguments,
3) initializing an instance of the *Netlist* class with the user design,
4) initializing the configuration for the stage,

5) initializing an instance of the *TargetData* class,

6) initializing an instance of the *NetlistConstraint* class,

7) executing the stage, and,

8) writing the netlist.

From a programming perspective, a stage uses the factory[1] design pattern for: 1) creating an instance of the selected algorithm (Algorithm Factory), and, 2) creating instances for the temporary data for a) a netlist (NetlistData Factory), b) a netlist node (NetlistNodeData Factory), and, c) a netlist edge (NetlistEdgeData Factory). The Algorithm Factory is located in the ***STAGE_PREFIX****AlgorithmFactory* class of the ***STAGE_NAME.****algorithm* package. The NetlistData Factory, NetlistNodeData Factory, and NetlistEdgeData Factory are located in the ***STAGE_PREFIX****NetlistDataFactory*, ***STAGE_PREFIX****NetlistNodeDataFactory* and ***STAGE_PREFIX****NetlistEdgeDataFactory* classes, respectively, of the ***STAGE_NAME.****algorithm.data* package. ***STAGE_PREFIX*** and ***STAGE_NAME*** are the **StagePrefix** and **StageName** fields of an *INFILE* shown in Figure 6.11 in Section 6.3. These files are generated by the framework, and, are updated when adding an algorithm to a project (Section 6.3).

14.2.1 Example

The following examples demonstrate the steps for adding the *logicSynthesis* stage as described in step 1) of Section 6.3.2.

1) Add the stage using the tools from the framework

   This step is detailed in step 1) of Section 6.3.2.

2) Add an attribute to the User Design

   This step is detailed in Section 11.1.

3) Define the result values in the *results.** JAVA package in the *src/results* folder

   For this step, the initial state of the project using the framework is shown in Figure 6.14.

1 https://en.wikipedia.org/wiki/Factory_method_pattern

```
...
public class LSResults {
        ...
        /**
         *  S_PRIMARYINPUT: value for LSResults
         */
        static public final String S_PRIMARYINPUT = "PRIMARY_INPUT";
        /**
         *  S_PRIMARYOUTPUT: value for LSResults
         */
        static public final String S_PRIMARYOUTPUT = "PRIMARY_OUTPUT";
        ...
}
```

Figure 14.4: Modifications to the *LSResults* class.

Figure 14.4 shows modifications to the *LSResults* class.

4) Implement the algorithm

This step is detailed in Section 14.3.1.

14.3 Algorithm

For an algorithm, the framework generates infrastructure files for: 1) implementing the algorithm, and, 2) storing temporary data within: a) a netlist, b) a netlist node, and, c) a netlist edge. The file for implementing the algorithm, ***ALGORIGHM_NAME***, is located in the ***ALGORIGHM_NAME*** class of the ***STAGE_NAME****.algorithm.****ALGORIGHM_NAME*** package**,** where ***STAGE_NAME*** is the **StageName** field of an *INFILE* shown in Figure 6.11 in Section 6.3**.** The location for storing temporary data for a) a netlist, b) a netlist node, and, c) a netlist edge of algorithm, ***ALGORIGHM_NAME***, are located in the ***ALGORIGHM_NAME****NetlistData*, ***ALGORIGHM_NAME****NetlistNodeData*, and, ***ALGORIGHM_NAME****NetlistEdgeData* classes, respectively, of the ***STAGE_NAME****.algorithm.****ALGORIGHM_NAME*** package**,** where ***STAGE_NAME*** is the **StageName** field of an *INFILE* shown in Figure 6.11 in Section 6.3**.**

The infrastructure file for implementing the algorithm, ***ALGORIGHM_NAME***, contains methods for accessing the instances of a ***ALGORIGHM_NAME****NetlistData*, a ***ALGORIGHM_NAME****NetlistNodeData*, and, ***ALGORIGHM_NAME****NetlistEdgeData*. These methods are *get****ALGORIGHM_NAME****NetlistNodeData(node)*, *get****ALGORIGHM_NAME****NetlistEdgeData(edge)*, and, *get****ALGORIGHM_NAME****NetlistData(netlist)*. Method *get****ALGORIGHM_NAME****NetlistNodeData(node)* requires an instance of the *NetlistNode* class, *node*, as a parameter, and, returns the ***ALGORIGHM_NAME****NetlistNodeData* instance attached to the *node*. Method *get****ALGORIGHM_NAME****NetlistEdgeData(edge)* requires an instance of the *NetlistEdge* class, *edge*, as a parameter, and, returns the ***ALGORIGHM_NAME****NetlistEdgeData* instance attached to the *edge*. Method *get****ALGORIGHM_NAME****NetlistData(netlist)* requires an instance of the *Netlist* class, *netlist*, as a parameter, and, returns the ***ALGORIGHM_NAME****NetlistData* instance attached to the *netlist*.

The infrastructure file for implementing the algorithm contains methods for logging during the execution of the algorithm. Logging with the Trace, Debug, Info, Warning, Error and Fatal log level are performed using the *logTrace(str)*, *logDebug(str)*, *logInfo(str)*, *logWarn(str)*, *logError(str)*, and, *logFatal(str)* methods respectively. The logger is defined by the *logger* member and accessed by the *getLogger()* method.

The infrastructure file for implementing the algorithm contains methods for accessing the parameters of the algorithm. These methods for accessing the parameters are: 1) *get**PARAMETER_NAME()*** for retrieving the parameter value, and, 2) *set**PARAMETER_NAME**(parameter)*** for setting the parameter value. ***PARAMETER_NAME*** refers to the name of the parameter as referenced in a parameter entry from the *INFILE* when the algorithm was added to the project (Section 6.3), and, *parameter* refers to the value of the parameter.

The purpose of an algorithm is to: 1) compute a result for an attribute in the netlist.

14.3.1 Example

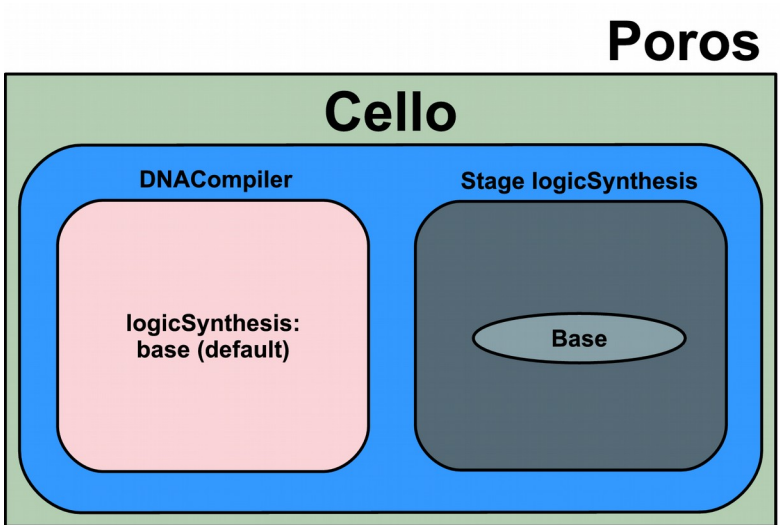

Figure 14.5: Initial state of the project using the framework

For the following examples, the initial state of the project using the framework is shown in Figure 14.5. The framework contains a project with name *Cello* that contains: 1) an application with name *DNACompiler*, and, 2) a stage with name *logicSynthesis* containing an algorithm with name *Base*. Note the initial state of the project shown in Figure 14.5 is identical to the state of the project after executing step 1) in Section 6.3.2, illustrated in Figure 6.15.

1) Implementing the *Base* algorithm

```
...
public class Base ... {
        ...
        @Override
        protected void getDataFromUCF() {
                <INSERT_CODE_FOR_RETRIEVING_DATA_FROM_TARGET_DATA>
        }
        ...
        @Override
        protected void validateParameterValues() {
                <INSERT_CODE_FOR_VALIDATING_PARAMETERS>
        }
        ...
        @Override
        protected void preprocessing() {
                <INSERT_CODE_FOR_PREPROCESSING>
        }
        ...
        @Override
        protected void run() {
                <INSERT_CODE_FOR_CORE_OF_ALGORITHM>
        }
        ...
        @Override
        protected void postprocessing() {
                <INSERT_CODE_FOR_POSTPROCESSING>
        }
        ...
}
```

Figure 14.6: Regions to insert the code for an algorithm. The region is highlighted by the tags in bold.

Figure 14.6 shows the regions to insert the code for an algorithm. The region is highlighted by the tags in bold. The ***<INSERT_CODE_FOR_RETRIEVING_DATA_FROM_TARGET_DATA>*** tag shows the location to insert the code for retrieving data from the *TargetData* instance. The ***<INSERT_CODE_FOR_VALIDATING_PARAMETERS>*** tag shows the location to insert the code for validating the parameters of the *Base* algorithm. The ***<INSERT_CODE_FOR_PREPROCESSING>*** tag shows the location to insert the code for preprocessing prior to executing the core of the *Base* algorithm. The ***<INSERT_CODE_FOR_CORE_OF_ALGORITHM>*** tag shows the location to insert the code for the core of the *Base* algorithm. The ***<INSERT_CODE_FOR_POSTPROCESSING>*** tag shows the location to insert the code for postprocessing subsequent to executing the core of the *Base* algorithm.

2) Storing temporary data for a) a netlist, b) a netlist node, and, c) a netlist edge of the *Base* algorithm

```
...
public class BaseNetlistData ... {
        ...
        <STORE_TEMPORARY_DATA>
        ...
}
```

Figure 14.7: Region to store temporary data for an instance of the *Netlist* class for the *Base* algorithm

Figure 14.7 shows the region to store temporary data for an instance of the *Netlist* class for the *Base* algorithm. The region is highlighted by the tag in bold. The ***<STORE_TEMPORARY_DATA>*** tag shows the location to insert the code for storing the temporary data.

```
...
public class BaseNetlistNodeData ... {
        ...
        <STORE_TEMPORARY_DATA>
        ...
}
```

Figure 14.8: Region to store temporary data for an instance of the *NetlistNode* class for the *Base* algorithm

Figure 14.8 shows the region to store temporary data for an instance of the *NetlistNode* class for the *Base* algorithm. The region is highlighted by the tag in bold. The ***<STORE_TEMPORARY_DATA>*** tag shows the location to insert the code for storing the temporary data.

```
...
public class BaseNetlistEdgeData ... {
        ...
        <STORE_TEMPORARY_DATA>
        ...
}
```

Figure 14.9: Region to store temporary data for an instance of the *NetlistEdge* class for the *Base* algorithm

Figure 14.9 shows the region to store temporary data for an instance of the *NetlistEdge* class for the *Base* algorithm. The region is highlighted by the tag in bold. The ***<STORE_TEMPORARY_DATA>*** tag shows the location to insert the code for storing the temporary data.

3) Accessing temporary data for a netlist node of the *Base* algorithm

```
...
public class Base ... {
	...
	@Override
	protected void run() {
		Netlist netlist = this.getNetlist();
		for (int i = 0; i < netlist.getNumVertex(); i++) {
			NetlistNode node = netlist.getVertexAtIdx(i);
			BaseNetlistNodeData nodeData = this.getBaseNetlistNodeData(node);
			<INSERT_CODE_FOR_PROCESSING_THE_DATA_IN_nodeData>
		}
	}
	...
}
```

Figure 14.10: Example for accessing the *BaseNetlistNodeData* instance of each *NetlistNode* instance within a *Netlist* instance.

Figure 14.10 shows an example for accessing the *BaseNetlistNodeData* instance of each *NetlistNode* instance within a *Netlist* instance. The ***<INSERT_CODE_FOR_PROCESSING_THE_DATA_IN_nodeData>*** tag shows the location for processing the data of the *nodeData* instance. The modification is show in bold.

4) Logging the execution of the *Base* algorithm

```
...
public class Base ... {
	...
	@Override
	protected void run() {
		this.logInfo("Executing run method");
	}
	...
}
```

Figure 14.11: Example for logging during the execution of the *Base* algorithm

Figure 14.11 shows an example for logging during the execution of the *Base* algorithm. The log level is *Info*. The modification is show in bold.

5) Accessing the value of the *BooleanType* parameter of the *Base* algorithm

```
...
public class Base ... {
        ...
        @Override
        protected void run() {
                boolean booleanTypeParam = this.getBooleanType();
                if (booleanTypeParam) {
                        ...
                }
                else {
                        ...
                }
        }
        ...
}
```

Figure 14.12: Example for accessing the value of the *BooleanType* parameter of the *Base* algorithm

Figure 14.12 shows an example for accessing the value of the *BooleanType* parameter of the *Base* algorithm. The modification is shown in bold.

14.4 Algorithm configuration file

The algorithm configuration file is a configuration file containing the algorithm parameters, their type and their default values. The file is in JavaScript Object Notation[1] (JSON) format.

14.4.1 Example

1 https://www.json.org/

```
{
	"name": "Base",
	"parameters":
	[
		{
			"name" : "BooleanType",
			"type" : "boolean",
			"value": true
		},{
			"name" : "ByteType",
			"type" : "byte",
			"value": 0
		},{
			"name" : "CharType",
			"type" : "char",
			"value": "c"
		},{
			"name" : "ShortType",
			"type" : "short",
			"value": 0
		},{
			"name" : "IntegerType",
			"type" : "int",
			"value": 0
		},{
			"name" : "LongType",
			"type" : "long",
			"value": 0
		},{
			"name" : "FloatType",
			"type" : "float",
			"value": 0.0
		},{
			"name" : "DoubleType",
			"type" : "double",
			"value": 0.0
		},{
			"name" : "StringType",
			"type" : "string",
			"value": "str"
		}
	]
}
```

Figure 14.13: Sample algorithm configuration file

AuthorName,*NewAuthor*,
ApplicationNames,*DNACompiler*,*other*
StagePrefix,*LS*,
StageName,*logicSynthesis*,
AlgorithmName,*Base*,
AlgorithmExtends,,
BooleanType,*TRUE*,boolean,
ByteType,*0*,byte,
CharType,*c*,char,
ShortType,*0*,short,
IntegerType,*0*,int,
LongType,*0*,long,
FloatType,*0.0*,float,
DoubleType,*0.0*,double,
StringType,*"str"*,string,

Figure 14.14: INFILE file driving the generation of the algorithm configuration file shown in Figure 14.13

Figure 14.13 shows a sample algorithm configuration file. Figure 14.14 shows the *INFILE* file driving the generation of the algorithm configuration file shown in Figure 14.13. For reference, the *INFILE* shown in Figure 14.14 is identical to that of the example corresponding to Figure 6.14.

The algorithm configuration file is represented by a JSON element. The element consists of a *name* attribute and a *parameters* attribute. The *name* attribute refers to the name of the algorithm. The *parameters* attribute contains the parameter(s) of the algorithm and is represented by a JSON array. Each element of the array represents a parameter. A parameter has three attributes: *name*, *type*, and, *value*. The *name* attribute refers to the name of the parameter, the *type* attribute refers to the type of the parameter, and, the *value* attribute refers to the default value of the parameter. In Figure 14.13: the *name* attribute of the algorithm is *Base*, and, the *parameters* attribute has nine elements. The first element has name: *BooleanType,* is of type: *boolean,* and, has default value: *true*. This element refers to the *BooleanType* parameter entry from the *INFILE* of Figure 14.14. Note the one-to-one mapping of the parameter entries defined in the *INFILE* of Figure 14.14 and the parameter elements from the algorithm configuration file from Figure 14.13.

14.5 Application configuration file

The application configuration file is a configuration file defining the default algorithm to execute when a stage is executed. The file is in JavaScript Object Notation[1] (JSON) format.

1 https://www.json.org/

14.5.1 Example

```
{
        "name": "DNACompiler",
        "stages": [
                {
                        "name": "logicSynthesis",
                        "algorithm_name": "Base"
                }
        ]
}
```

Figure 14.15: Sample application configuration file for the *DNACompiler* application

Figure 14.15 shows a sample application configuration file for the *DNACompiler* application. Figure 14.14 shows the *INFILE* file driving the addition of the configuration for the *logicSynthesis* stage shown in Figure 14.15. For reference, the *INFILE* shown in Figure 14.14 is identical to that of the example corresponding to Figure 6.14.

The application configuration file is represented by a JSON element. The element consists of a *name* attribute and a *stages* attribute. The *name* attribute refers to the name of the application. The *stages* attribute contains the stage configuration(s) of the application and is represented by a JSON array. Each element of the array represents a stage configuration. A stage configuration has two attributes: *name*, and, *algorithm_name*. The *name* attribute refers to the name of the stage instance, and, the *algorithm_name* attribute refers to the name of the default algorithm to execute when the stage is executed. In Figure 14.15: the *name* attribute of the application is *DNACompiler*, and, the *stages* attribute has one element. The element has name: *logicSynthesis,* with algorithm_name: *Base*.

Manually editing the application configuration file

There are three use cases for manually editing the application configuration file: 1) modifying the default algorithm for a stage without using the execution control file, 2) enabling a default algorithm for a stage instance when a stage exists within a project but the default algorithm has been removed (as seen in Section 6.4.2), and, 3) adding a new instance of a stage where the infrastructure for the stage exists within the project.

1) To modify a default algorithm for a stage without using the execution control file, override the the value of the *algorithm_name* attribute for the target stage.

```
{
        "name": "DNACompiler",
        "stages": [
                {
                        "name": "logicSynthesis",
                        "algorithm_name": "NewBase"
                }
        ]
}
```

Figure 14.16: The modification to the configuration file shown in Figure 14.15 for changing the default algorithm of the *logicSynthesis* stage without using the execution control file

```
logicSynthesis,NewBase
```

Figure 14.17: An equivalent execution control file to the modifications shown in Figure 14.16

Figure 14.16 shows the modification to the configuration file shown in Figure 14.15 for changing the default algorithm of the *logicSynthesis* stage without using the execution control file. The modification is shown in bold. *NewBase* is the default algorithm for the *logicSynthesis* stage. Figure 14.16 shows an equivalent execution control file to the modification shown in Figure 14.16. For reference, the execution control file shown in Figure 14.17 is identical to that of the example corresponding to Figure 9.3.

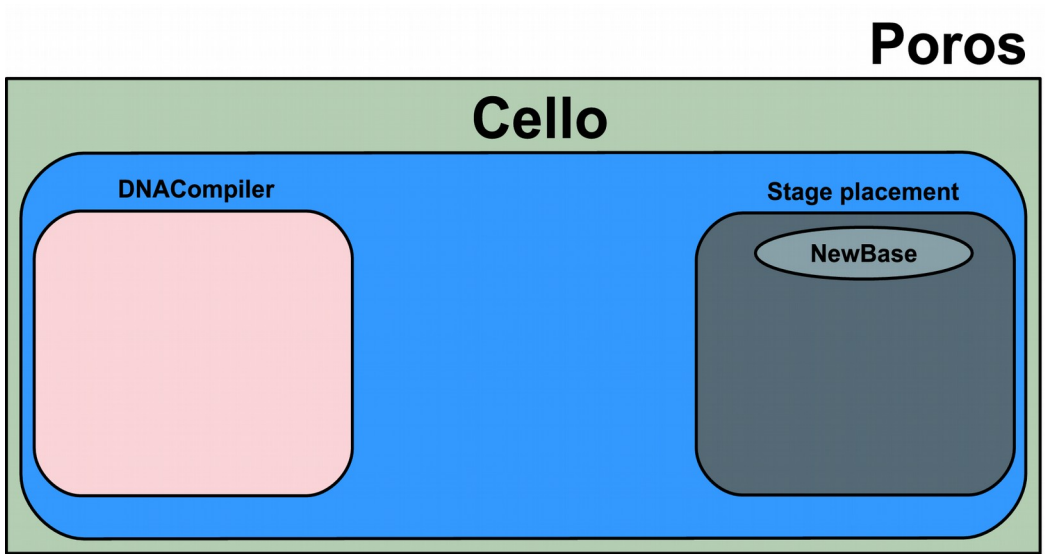

Figure 14.18: The initial state of the project using the Poros framework

```
{
        "name": "DNACompiler",
        "stages": [
        ]
}
```

Figure 14.19: The initial state of the *DNACompiler* application configuration file of the project using the Poros framework shown in Figure 14.18

2) To enable a default algorithm for a stage instance when a stage exists within a project but the default algorithm has been removed (as seen in Section 6.4.2), add an element to the stages attribute. For the following example, the initial state of the project using the Poros framework is shown in Figure 14.18. The framework contains a project name *Cello*. The *Cello* project contain an application named *DNACompiler*, and, a stage named *placement* with algorithm *NewBase*. For reference, the state of the project is identical to the example in Section 6.4.2 after executing step 4) as shown in Figure 6.27. Figure 14.19 shows the initial state of the *DNACompiler* application configuration file of the project using the Poros framework shown in Figure 14.18.

```
{
        "name": "DNACompiler",
        "stages": [
                {
                        "name": "placement",
                        "algorithm_name": "NewBase"
                }
        ]
}
```

Figure 14.20: The modification to the configuration file shown in Figure 14.19 for enabling a default algorithm for a stage instance when a stage exists within a project but the default algorithm has been removed (as seen in Section 6.4.2)

Figure 14.20 shows the modification to the configuration file of Figure 14.19 for enabling a default algorithm for a stage instance when a stage exists within a project but the default algorithm has been removed (as seen in Section 6.4.2). The modification is shown in bold. *NewBase* is the default algorithm for the *placement* stage.

3) To add a new instance of a stage where the infrastructure for the stage exists within the project, add an element to the stages attribute with a unique value for the *name* attribute. A unique value signifies that no other entry within the array contains the given value for the *name* attribute. The application will need to execute the added stage instance.

```
{
        "name": "DNACompiler",
        "stages": [
                {
                        "name": "placement",
                        "algorithm_name": "NewBase"
                },
                {
                        "name": "placementOther",
                        "algorithm_name": "NewBase"
                }
        ]
}
```

Figure 14.21: Modification to the configuration file shown in Figure 14.20 for adding a new instance of a stage where the infrastructure for the stage exists within the project

Figure 14.21 shows the modification to the configuration file shown in Figure 14.20 for adding a new instance of a stage where the infrastructure for the stage exists within the project. The name of the new instance of the stage is *placementOther*, its default algorithm is *NewBase*. The modification is shown in bold.

```
...
public class Main {
        public static void main(String[] args) {

                ...
                /*
                 * Add Stages below
                 */
                // placement
                currentStage = appCfg.getStageByName("placement");
                PLRuntimeObject PL = new PLRuntimeObject(currentStage, td, netlist, runEnv);
                PL.execute();
                // placementOther
                currentStage = appCfg.getStageByName("placementOther");
                PLRuntimeObject PL0 = new PLRuntimeObject(currentStage, td, netlist,
runEnv);
                PL0.execute();
                /*
                 * Add Stages above
                 */
                ...
        }
}
```

Figure 14.22: Modification to the *main* method of the *Main* class of an application for executing a new instance of the *placement* stage, *placementOther*

Figure 14.22 shows the modification to the *main* method of the *Main* class of an application for executing a new instance of the *placement* stage, *placementOther*. The modification is shown in bold. The first instance of the *placement* stage is *PL*, and, the second instance to the *placement* stage is *PL0*.

15 Programming Tutorial: Advanced Use Cases

This chapter shows examples for advanced use cases for programming in the framework.

15.1 Examples

Executing a python script in a project using Poros

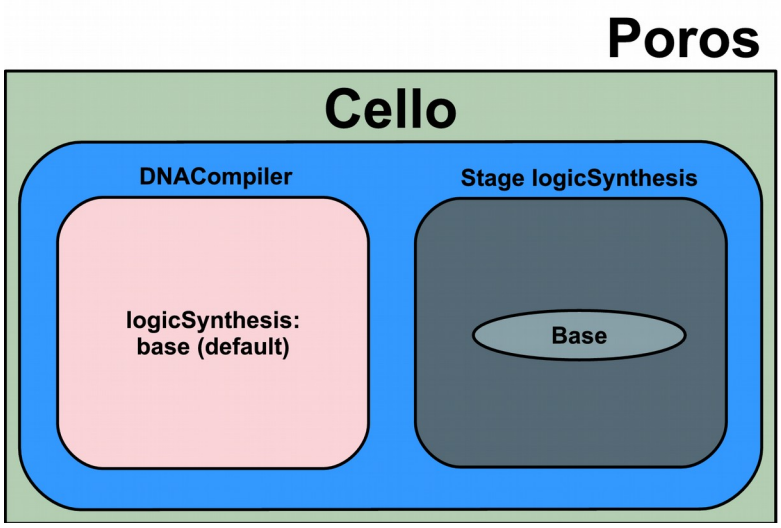

Figure 15.1: Initial state of the project using the framework

For the following example, the initial state of the project using the framework is shown in Figure 15.1. The framework contains a project with name *Cello* that contains: 1) an application with name *DNACompiler*, and, 2) a stage with name *logicSynthesis* containing an algorithm with name *Base*. Note the initial state of the project shown in Figure 15.1 is identical to the state of the project after executing step 1) in Section 6.3.2, illustrated in Figure 6.15. The name of the python script is *gates.py*. This script is used in the *Base* algorithm.

1) Place the file containing the python script in resource directory of the *logicSynthesis* stage

Create the gates folder in the *Cello/src/resources-logicSynthesis/executable/scripts/* directory. Copy *gates.py* to the *Cello/src/resources-logicSynthesis/executable/scripts/gates/* directory.

2) Define an attribute for storing the reference to *gates.py* and the methods for accessing the attribute

```
public class Base ... {
        ...
        protected void setPythonScriptFilename(final String str) {
                this.pythonScriptFilename = str;
        }
        protected String getPythonScriptFilename() {
                return this.pythonScriptFilename;
        }
        private String pythonScriptFilename;
        ...
}
```

Figure 15.2: Definition of attribute, *pythonScriptFilename*, for storing the reference to *gates.py* and the methods for accessing the attribute.

Figure 15.2 shows the definition of attribute, *pythonScriptFilename*, for storing the reference to *gates.py* and the methods for accessing the attribute. The modification is shown in bold.

3) Define the reference to *gates.py* in the preprocessing method

```
import common.Utils;
...
import logicSynthesis.common.LSUtils;

public class Base ... {
        ...
        @Override
        protected void preprocessing() {
                // exec
                String exec = "";
                exec += LSUtils.getResourcesFilepath();
                exec += Utils.getFileSeparator();
                exec += "executable";
                exec += Utils.getFileSeparator();
                exec += "scripts";
                exec += Utils.getFileSeparator();
                exec += "gates";
                exec += Utils.getFileSeparator();
                exec += "gates.py ";
                this.setPythonScriptFilename(exec);
        }
        ...
}
```

Figure 15.3: Definition for referencing to *gates.py* in the *preprocessing* method.

Figure 15.3 shows the definition for referencing to *gates.py* in the *preprocessing* method. The modification is shown in bold.

4) Execute *gates.py*

```
import logicSynthesis.runtime.environment.LSArgString;
...
public class Base ... {
        ...
        @Override
        protected void run() {
                String cmd = "";
                cmd += this.getRuntimeEnv().getOptionValue(LSArgString.PYTHONENV);
                cmd += " ";
                cmd += this.getPythonScriptFilename();
                Utils.executeAndWaitForCommand(cmd);
        }
        ...
}
```

Figure 15.4: Execute *gates.py* in the *run* method.

Figure 15.3 shows the execution of *gates.py* in the *run* method. The modification is shown in bold.

Using an external Java library (JAR file) in a project using Poros

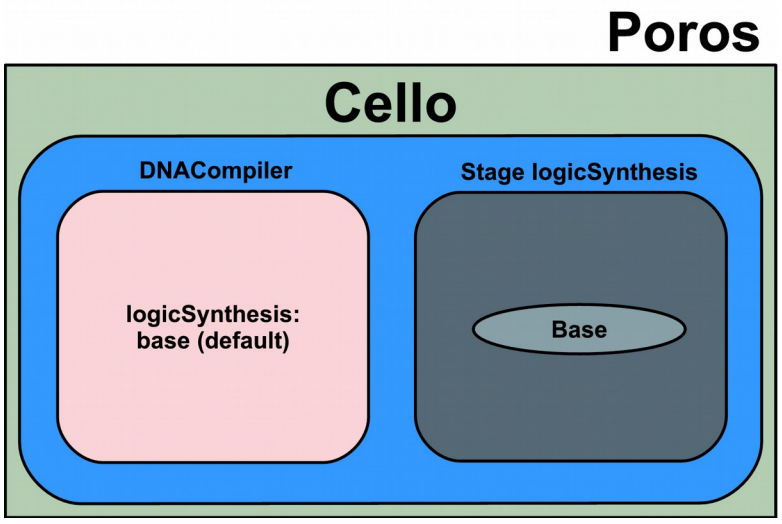

Figure 15.5: Initial state of the project using the framework

For the following example, the initial state of the project using the framework is shown in Figure 15.5.The framework contains a project with name *Cello* that contains: 1) an application with name *DNACompiler*, and, 2) a stage with name *logicSynthesis* containing an algorithm with name *Base*. Note the initial state of the project shown in Figure 15.5 is identical to the state of the project after executing step 1) in Section 6.3.2, illustrated in Figure 6.15. The name of the JAR file is *gates.jar*. This JAR file is used in the *Base* algorithm.

1) Place JAR file in resource directory of the *Base* algorithm of the *logicSynthesis* stage

Copy *gates.jar* to the *Cello/src/resources-logicSynthesis/algorithms/Base/* directory.

2) Update the User Library by adding the external Jar to the Project

1. Click on Windows > Preferences
2. Java > Build Path > User Libraries
3. Select 'PorosCompilerlib'
4. Click Add Jars...
5. Select the *gates.jar* in the *Cello/src/resources-logicSynthesis/algorithms/Base/* directory

3) Use the class(es) from the Java library (Jar file)

```
import gates.datastructure.Gate;
...
public class Base ... {
        ...
        @Override
        protected void run() {
                List<Gate> gates = new ArrayList<Gate>;
                for (int i = 0; i < this.getNetlist().getNumVertex(); i++) {
                        gates.add(new Gate());
                }
                ...
        }
        ...
}
```

Figure 15.6: Use the *Gate* class from the *gates.jar* Java library

Figure 15.6 shows an example of using the *Gate* class from the *gates.jar* Java library. The modification is shown in bold.