



NeuBiro
Programmer's guide

2017-09-14

Version 0.7

Table of contents

1. Introduction	1
2. Basic concepts	2
3. The input dataset	3
4. The import specifications	4
4.1. Master table definition	5
4.2. Context variables definition	6
4.3. Fields definition	6
4.4. Calculated fields definition	8
4.5. Lookup tables	11
4.5.1. Using lookups tables from calculated fields	14
4.6. Quality checks	15
4.6.1. Records deduplication	17
4.7. Accessing the internal database	18
5. The statistical packages	20
5.1. The module's descriptor	21
5.2. The select unit specifications	25
5.3. The module's statistical code	26
5.3.1. Parameters passed to the R code from NeuBiro	29
5.3.2. Creation of zip compressed files	30
5.4. The report template	33
6. Example statistical package	34
6.1. The sample dataset	34
6.2. The sample import.specs	34
6.3. The sample statistical package	35
7. Hacking NeuBiro	36
7.1. Accessing the source code	36
7.2. Building NeuBiro	36
7.3. Documentation	37
8. References	38
9. Acknowledgements	39

1. Introduction

In this guide are explained all the concepts behind the NeuBiro's configuration.

The configuration files are based on a simple DSL (Domain Specific Language) to provide a good degree of customization.

As described in the user's guide NeuBiro is agnostic about the data provided for the analysis and every aspect of the analysis must be described via configuration files.

The next image shows an overview of the working pipeline of NeuBiro.

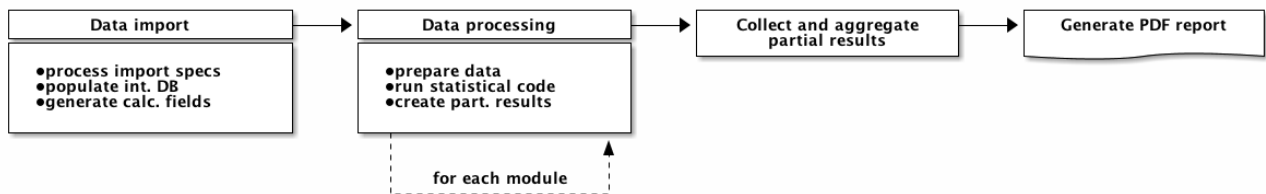


Figure 1. overview of the main workflow

2. Basic concepts

The main components needed to create an analysis with NeuBiro are:

- the input dataset
- the import specifications
- the statistical package
- the report template

All the configuration files are based on a simple DSL which uses the Groovy syntax.

TBW

3. The input dataset

Everything starts with the data; NeuBiro is capable of reading an input stream in CSV format.

For the process to succeed it is important that the CSV file has an header representing the field names:

```
NAME,SURNAME,BIRTH_DATE  
John,Doe,15/03/1970  
Mary,Doe,24/10/1970  
...  
Michael,Smith,10/01/1968
```

In this way the importer module, can recognize every single column with a unique name. To make its work, the importer module, relies on the import specifications as we will see in the next section.

The import process reads the entire file and creates internal tables backed by a relational database.

4. The import specifications

The import specifications are the way we have to map the input data. Only the fields defined in the specifications will be imported. This way we can have a big dataset that can be reused for different analysis and it is possible to optimize our code and memory usage using only the fields we need.

The entire import process is summarized in the following image:

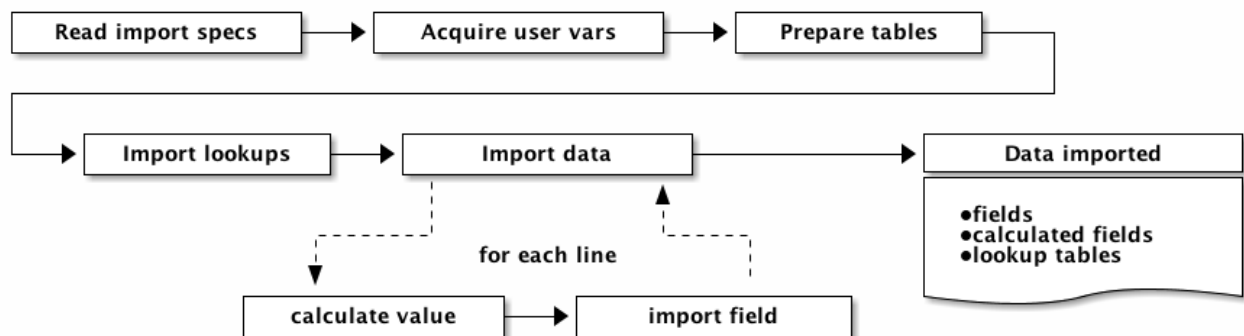


Figure 2. Overview of the import process

What follows is an example for an import specification:

```

master { ①
  'THETABLE' { ②
    context { ③
      ...
    }
    fields { ④
      ...
    }
    calculatedFields { ⑤
      ...
    }

    recordCheck = { ⑥
      ...
    }

    locf { ⑦
      ...
    }
  }
}
  
```

- ① master block
- ② master table block
- ③ variables definition

- ④ fields definitions
- ⑤ calculated fields definitions
- ⑥ recordCheck definition
- ⑦ LOCF configuration



Even if the DSL permits the definition of multiple tables only one master table is allowed in the actual NeuBiro implementation

4.1. Master table definition

The master table definition describes the input dataset; all the definitions are enclosed in a `master {}` block

Inside the `master {}` block can be used the following elements:

Option name	Type	Default	Notes
label	string	false	The table description
mandatory	boolean	true	The table is mandatory (table in master block is always mandatory)
context {}	—	—	Definition of the variables to request to the user
fields {}	—	—	Definition of the fields to import from the dataset
calculatedFields {}	—	—	Definition of the calculated fields
recordCheck = {}	code	—	Record level checks
locf {}	—	—	Configuration for the LOCF algorithm

The following example code defines a master input table named `PERSONS_TABLE`

```

master {
  'PERSONS_TABLE' { ①
    label = "Main persons table" ②
    mandatory = true ③
  }
}

```

- ① name of the table when imported in the internal database
- ② description of the table
- ③ this table is mandatory



Master table is always mandatory and must be specified by the user before starting the import. This flag is useful when using optional lookup tables as we will see in the next sections.

Has to be noted that, even if the syntax of the `master {}` permits to specify more tables the current NeuBiro implementation uses **only one** table.

4.2. Context variables definition

Context variables can be used to ask information to the user that can be useful to the import process.

These variables can be defined in the `context {}` block:

```
context {
  'CUR_YEAR' {
    type = "integer"
    label = "Current year"
    mandatory = true
  }
}
```

And the following options are available:

Option name	Type	Default	Notes
type	string	string	one of integer, decimal, string
label	string	—	The variable's description
mandatory	boolean	false	The table is mandatory

The variable defined here will populate an input area on the import tab and can be used in calculated fields using the `context` associative array.

4.3. Fields definition

Inside the table block we have to define the fields we want to map (and import). Only the fields specified in this block will be used to create the internal database table.

To define the fields we need a `fields {}` block, the following elements are available:

Option name	Value	Default	Notes
type	see table	varchar	The type of the field
size	numeric	255	The size of the field in case of a varchar
format	string	—	The format used to parse the field (eg. yyyy/MM/dd)
nameTo	string	—	The name to assign to the field in the internal database
valid	code	—	Code block that returns true if the field's value is valid

Option name	Value	Default	Notes
mandatory	boolean	false	The field is mandatory (eg. cannot be missing otherwise the entire record is discarded)

The possible values of the `type` option can be:

Type	Description
varchar	alphanumeric characters
int	integer value
smallint	small integer value
boolean	boolean value
date	date value

Fields can be defined as follow:

```
fields {
  'FIELD_NAME_1' { ①
    type = "varchar" ②
    size = 10 ③
    nameTo = "NEW_FIELDNAME_1" ④
  }

  'FIELD_NAME_2' {
    type = "int"
  }
  ...
}
```

- ① the quoted name before the opening bracket is the field name as defined in the CSV file (`FIELD_NAME_1` in the example).
- ② type of the field
- ③ size of the field (used only for varchar)
- ④ the name we want to use in the internal database

The field's name has to match the one used in the dataset file we want to import.

The `fields {}` block must reside into the table block:

```

master {
  'PERSONS_TABLE' {
    ...
    fields {
      'FIELD_NAME_1' {
        type = "varchar"
        size = 10
        nameTo = "NEW_FIELDNAME_1"
      }
    }
  }
}

```

At the end, in the above example, we are creating an internal table named `PERSONS_TABLE` and, after the import, the table will have **one** field named `NEW_FIELDNAME_1` of type `varchar` and size `10` the value of the field will be the one named `FIELD_NAME_1` from the CSV dataset.

The fields will be imported in the internal database after the conversion to the specified type. In case of fields having a particular format (eg. dates) it is important to also specify the `format = ""` option (eg. `yyyy/MM/dd`) otherwise the field will not be parsed correctly hence it will result in a missing value imported into the database.

For each field can also be specified a `valid {}` block:

```

fields {
  'FIELD_1' {
    type = "int"

    valid = { value -> ①
      if (value < 10 && value > 20) {
        false
      }
    }
  }
}

```

① the code block used to check if field is valid

The optional `valid` code block receives the actual value of the field and therefore can perform all the checks needed to validate it. If the code block returns `false` the value of the field **will be set to missing** otherwise it will remain untouched.

A typical use of this feature is to enforce valid ranges for the field.

4.4. Calculated fields definition

If needed we can also define **calculated fields**; a calculate field is a field not originally present into the dataset that will be created on the fly during the import process.

The `calculatedFields {}` must reside inside the table block, at the same level of the `fields {}` block:

```

master {
  'PERSONS_TABLE' {
    ...
    fields {
      ...
    }
    calculatedFields {

    }
  }
}

```

This kind of fields are defined inside a `calculatedFields {}` block and the available options are:

Option name	Value	Default	Notes
type	see table	varchar	The type of the field
size	numeric	255	The size of the field in case of a varchar
persist	boolean	true	Flag to indicate if the field must create a column in the internal table or not
value	code	—	Code block that calculates the field's value
mandatory	boolean	false	The field is mandatory (eg. cannot be missing otherwise the entire record is discarded)

What follows is an example definition of a calculated field:

```

calculatedFields {
  'CALC_FIELD_1' { ①
    type = "varchar" ②
    size = 10 ③
    persist = false ④
    value = { record, context -> ⑤
      "A VALUE" ⑥
    }
  }
}

```

- ① the quoted name before the opening bracket is the field name
- ② type of the field
- ③ size of the field (used only for varchar)
- ④ flag to persist or not the field into the internal table
- ⑤ the code block used to populate the field's value

⑥ in the Groovy language the return statement can be omitted

The `persist` option is useful when we want to split the calculation of a field in several steps, using `persist = false` we are creating a temporary variable that will not be written in the internal database.

The `value {}` block represents the way we have to produce the value for our calculated field. The code block accepts 2 arguments:

Arg name	Description
record	associative array containing all the fields imported from the input dataset
context	associative array containing values entered by the user as defined in the <code>context {}</code> block of the import specifications

and return the value that will be used to populate the field.

In the `value {}` block we can use all the standard libraries provided by Groovy/Java, an example code can be:

```

calculatedFields {
  'RECORD_DATE' {
    persist = false ①
    type = "date"
    value = { record, context ->
      try {
        Date.parse("yyyy-MM-dd", record['BIRTH_DATE']) ②
      }
      catch(Exception) {
        null
      }
    }
  }
}

```

① only internal use, we don't want this field into the final table

② Java/Groovy code to parse the textual date

The above example creates a new field called `RECORD_DATE` of type `date` parsing the string representation of the field `BIRTH_DATE` coming from the input dataset.

A more useful example can be the following:

```

'BIRTH_DATE' { ①
  persist = false
  type = "date"
  value = { record, context ->
    try {
      Date.parse("yyyy-MM-dd", record['BIRTHDAY'])
    }
    catch(Exception) {
      null
    }
  }
}

'AGE' { ②
  type = "int"
  value = { record, context -> //
    start = record["BIRTH_DATE"]
    recordDate = record["RECORD_DATE"]
    yearsBetween = recordDate[Calendar.YEAR] - start[Calendar.YEAR] ③
  }
}

```

- ① temporary fields to store the birth date in Date format
- ② the `AGE` field will be persisted and will contain the age in years
- ③ return value (in Groovy the last expression evaluated)

Once a calculated field is created it is inserted into the `record` associative array; in this way its value will be available for processing by other calc fields. It is important to note that the order in which the calc fields appears in the source code is significant; we cannot get the value of a field not already calculated.

This is demonstrated in the above example where we first create a field named `BIRTH_DATE` and then, in the next calc field, we use it.



Please note that the field's name used to retrieve a value (eg. `record['FIELDNAME']`) must be the one as defined in the input file even if we use the `nameTo` to rename it.

4.5. Lookup tables

The import specifications can define lookup tables as well. Such tables can be used to integrate the master dataset and/or be used to create additional fields into the internal database of NeuBiro.

These tables are external to the master dataset and are imported in a second step; in reality the lookup tables are imported **before** the master dataset. In this way we can have access to all the information contained in them during the creation of calculated fields.

Lookup tables are defined inside a `lookups {}` block and their definition is very similar to the one used to describe the master table.

Inside the `lookups {}` block can be used the following elements:

Option name	Type	Default	Notes
label	string	false	The table description
skipAutoId	boolean	false	Flag to indicate if the import process must skip the creation of the <code>ID</code> field
mandatory	boolean	false	The table is mandatory
fields {}	—	—	Definition of the fields to import from the dataset
calculatedFields {}	—	—	Definition of the calculated fields
indexes {}	—	—	Definition of the indexes to create for the lookup table

In the `lookups {}` we can define as many tables as we need.

An example definition can be:

```
lookups {
  'LOOKUP_TABLE' { ①
    label = "Description"
    mandatory = true ②
    skipAutoId = true ③
    fields {
      ... ④
    }

    calculatedFields {
      ... ⑤
    }

    indexes {
      ... ⑥
    }
  }
}
```

- ① the name of the lookup table
- ② the table is mandatory and must be imported
- ③ do not create an ID field
- ④ fields definitions
- ⑤ calculated fields definitions
- ⑥ indexes definitions

The structure and the format of this block is the same already seen for the master table but with some additions in the available config options:

- the `skipAutoId` option
- the `indexes {}` block

The `skipAutoId` option indicates to the importer that an ID column must not be created; if not specified or assigned with `false` the resulting table in the internal database will have an ID column with a unique value.

The `indexes {}` block permits the creation of indexes on the lookup table and are mainly used to improve the performance of the system.

Inside the `indexes {}` block can be used the following options:

Option name	Type	Default	Notes
primary	boolean	true	Flag to indicate that the index is primary (imply unique)
unique	boolean	false	Flag to indicate that the index is unique (no repeated keys allowed)
fields	string's list	—	List of field's names that composes the index

In the following example we will define a lookup table for a codification table:

```
lookups {
  'CODES' { ①
    label = "Internal codes"
    mandatory = true ②
    skipAutoId = true ③
    fields {
      'CODE' {
        type = "varchar"
        size = 10
      }
      'DESCRIPTION' {
        type = "varchar"
        size = 20
      }
    }
  }

  indexes { ④
    'codeidx' { ⑤
      primary = true ⑥
      fields = ['CODE'] ⑦
    }
  }
}
}
```

① name of the lookup table

- ② the table must be imported
- ③ do not create the ID field (we will use our own index)
- ④ indexes definition
- ⑤ the name of the index is `codeidx`
- ⑥ the index is primary
- ⑦ the index is based on the field named `CODE`

4.5.1. Using lookups tables from calculated fields

Once the lookup tables are imported we need a way to retrieve the data. The following code shows this technique:

```

1 'DECODED_FIELD' { ①
2   type = "varchar"
3   size = 100
4   value = { record, context ->
5     def code = record['CODE']?.trim()
6     if (code) {
7       def codes = mmg.lookup("CODES", "CODE", ["CODE", "DESCRIPTION"]) ②
8       if (rows) {
9         def result = codes[0].CODE
10        result
11      }
12      else {
13        null
14      }
15    }
16    else {
17      null
18    }
19  }
20 }

```

- ① calculated field definition
- ② use of the lookup function

In the example we see a normal definition of the calculated field but, at line 7, <2> we can note the usage of the `lookup()` function.

The syntax of the function is:

```
OBJ.lookup(LOOKUP_TABLE, FIELD_TO_SEARCH, LIST_OF_FIELDS_TO_RETRIEVE)
```

where:

- `OBJ` is the string (literal or a variable) containing the value we want to lookup.

- `FIELD_TO_SEARCH` is the name of the field in the lookup table in which to find the value specified in `OBJ`
- `LIST_OF_FIELDS_TO_RETRIEVE` is the list of fields name to retrieve from the selected records of the lookup table

The function returns an array, each element of the array is a map containing the fields specified in `LIST_OF_FIELDS_TO_RETRIEVE`

By example if we want to retrieve the description associated to the code `009` from the lookup tables named `CODES`

```
def codeToSearch = "009"
def values = codeToSearch.lookup("CODES", "CODE", ["DESCRIPTION"])

def description = values[0].DESCRIPTION
```

The variable `description` will contain the decode value.

4.6. Quality checks

Data quality check within the import procedure is essential:

- to make the data custodian aware of any pitfalls in data collection, storage or exchange formats
- to discover errors in input data or the configuration of import (selection of wrong units of measurement for numeric fields, wrong values for enumerated fields, wrong format for date fields)
- to reduce the risk of failure or unexpected behavior during statistical processing
- to reduce the risk of biased results in any statistical output

In NeuBiro the quality checks are implemented into the `import.specs` file and can be differentiated into 2 levels:

- field level
- record level

For the field level there are 2 types of checks: implicit and explicit.

Implicit checks are performed by the import engine with the goal to set to missing a field's value if it doesn't respect the format defined in the specs. The explicit checks are written by the user using the `valid = {}` block in the field's definition. The valid check should return a true or false value, if false is returned the field's value will be set to missing.

The record level check can "observe" the entire record imported and with all calculated fields already in place. At this level the code can update field's values, perform checks record wide and even choose to discard the record entirely.

For example the record check can be used to verify the out of range values for specific fields or to

set the value of one field depending on the values of others in the same record.

The record check is defined in the following way:

```

1 master {
2   'MASTERTABLE' {
3     ...
4     recordCheck = { record ->
5       def newRecord = record ①
6       def action = "SAVE"
7       def message
8
9       if (newRecord['DOB'] > newRecord['EPI_DATE']) {
10        action = "DISCARD" ②
11        message = "The record has incoherent values and will be discarded"
12      }
13
14      if (newRecord['AGE'] < 0) {
15        newRecord['AGE'] = null ③
16      }
17
18      return [ ④
19        action: action,
20        message: message,
21        record: newRecord
22      ]
23    }
24    ...
25    fields {
26      ...
27    }
28    calculatedFields {
29
30    }
31  }
32 }

```

- ① copy the original record in a temporary variable
- ② record will be discarded if dates are wrong
- ③ set the AGE field to missing
- ④ the return structure

The return value of the `recordCheck` code is an associative array and must have the following structure:

```

1 [
2   action: action,    ①
3   message: message, ②
4   record: newRecord ③
5 ]

```

- ① what to do with the record, options are SAVE or DISCARD
- ② If DISCARD this message will be shown in the log
- ③ the record to be written

The code of the `recordCheck` block will be evaluated for each imported row. The returned record will be written into the internal database **but only after the check of the mandatory fields**; in fact if a mandatory field is empty the record will be always discarded.

4.6.1. Records deduplication

NeuBiro provides a simple LOCF (Last Observation Carried Forward) algorithm implementation.

The behaviour of the algorithm can be controlled adding a block named `locf {}`, inside the `locf {}` block can be used the following options:

Option name	Type	Default	Notes
keys	string's list	—	List of field's names that composes the unique key
order	string's list	—	List of fields to order the table with
exclude	string's list	—	List of fields whose values will not carried forward

The block for the LOCF task can be defined as follow:

```

master {
  'MASTERTABLE' {
    ...
    locf { ①
      table = "MASTER_LOCF" ②

      keys = ['PAT_ID'] ③
      order = ['PAT_ID', 'EPI_DATE'] ④
      exclude = ['CHOL', 'HDL', 'LDL'] ⑤
    }
  }
  ...
}

```

- ① locf block
- ② the name of the table that will be created after the LOCF task
- ③ aggregate by PAT_ID
- ④ Sort by PAT_ID, EPI_DATE
- ⑤ do not carry forward on CHOL, HDL and LDL

At the end of the process a new table (named as defined in **table**) will be created into the internal database, such a table will be compacted to just one record per unique key.

Please note that the fields specified in **order** are very important for the process to succeed.

4.7. Accessing the internal database

The internal database used by NeuBiro is a standard SQL database; when NeuBiro is running it exports an endpoint that can be used to explore all the tables stored into the system.

To start the database console we can use a customized launch script; from the command line:


Launch the db console from Windows

```
cd <NEUBIRO_INSTALL_ROOT>  
bin\dbconsole.bat
```


Launch the db console from linux/OS X


```
cd <NEUBIRO_INSTALL_ROOT>  
bin/dbconsole
```

The above command will open the default internet browser on the connection page of the console:

English  Preferences Tools Help

Login

Saved Settings: 

Setting Name:  Save Remove

Driver Class:

JDBC URL:

User Name:


Password: 

Figure 3. H2 database console connection window

The **JDBC URL** required by the console is as follow:

```
jdbc:h2:tcp://localhost:9123/file:///FULL_PATH_TO_DB_FILE/neubiro-tcp-db
```

where **FULL_PATH_TO_DB_FILE** is the installation path of NeuBiro. The fields **User Name** and **Password** must be left blank.

Please keep in mind that for this to work NeuBiro **must** be running.



For more information about the db console please refer to the H2 database documentation at the following url:
<http://www.h2database.com/html/tutorial.html>

5. The statistical packages

The statistical package is the core of the analysis and it is composed by, at least, one module.

Each module is in turn composed by a specification file `indicator.specs` and one (or more) file written in `R language`.

Usually the statistical package also contains the import specifications described in the previous chapter.

A typical structure of a statistical package is:

```
<ROOT>
+-- import
|   |-- import.specs ①
|-- modules ②
|   +-- module_1
|   |   +-- indicator.specs ③
|   |   |-- indicator.r ④
|   +-- module_2
|   |   +-- indicator.specs
|   |   |-- indicator.r
|   +-- ...
|   +-- module_n
|   |   +-- indicator.specs
|   |   |-- indicator.r
|   +-- report ⑤
|   |   +-- master.xml ⑥
|   |   +-- chapter.xml ⑦
|   |   +-- master.xsl ⑧
|   |   |-- resources ⑨
|   |   |-- logo.png
|   |-- selectUnit.specs ⑩
```

- ① import specification
- ② modules root
- ③ module descriptor
- ④ entry point for the module code
- ⑤ final report descriptor's root
- ⑥ master file for final report
- ⑦ master file for each chapter of the final report
- ⑧ master xsl file for final report formatting
- ⑨ additional resources
- ⑩ select unit specs

Due to the fact that the data and the statistical code who process them are tightly coupled and is better to keep them together but this is not mandatory.

With these information NeuBiro, can implement the following execution schema:

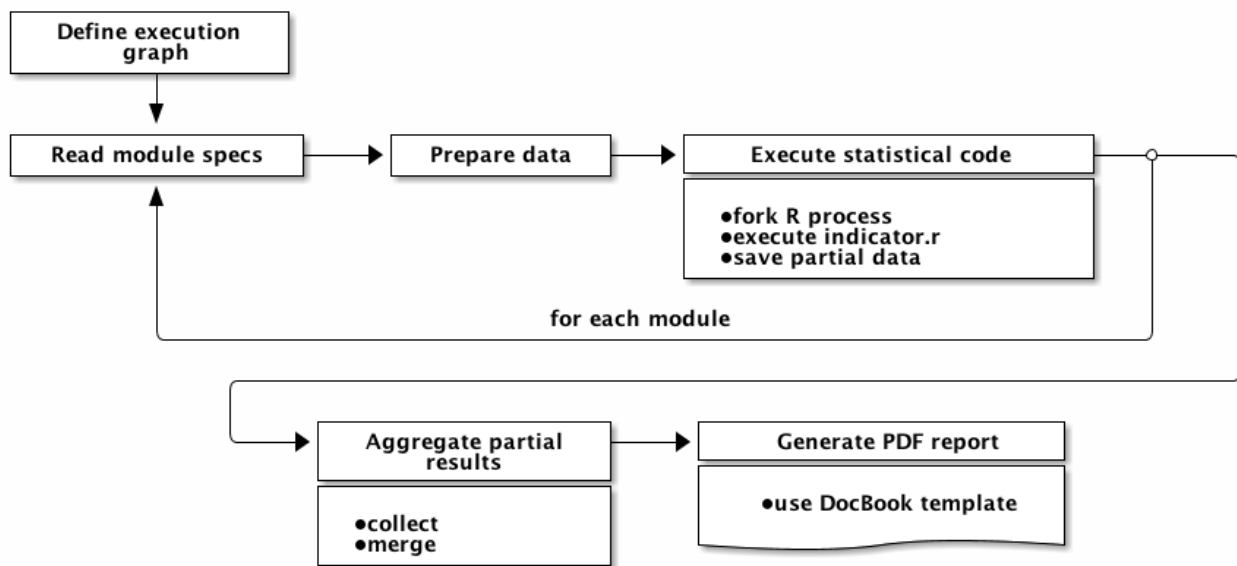


Figure 4. Overview for the execution phase

Essentially NeuBiro creates the execution graph taking care of the dependencies between modules and for each module performs the following tasks:

- the module's data are prepared (as defined in the module's spec)
- the R process is invoked and writes on disk the partial data xml data and other artifacts eg images
- the R process returns and NeuBiro performs a check to verify the data have been created

When all the the modules have been executed all the partial outputs are merged together to create the final report.

5.1. The module's descriptor

A module descriptor describes the preparation of the data needed to the companion R code; NeuBiro will use this information to prepare all the needed artifacts cthat in turn will be referenced from the statistical code.

The goal of this approach is that each module can concentrate on a small part of the computation using only the data it needs optimizing code and resources. Each module can depend from other modules so we can, by example, an hidden module who produces common artifacts that can be reused by others.

A module descriptor must reside in a file named `indicator.specs` and contains a single `indicator {}` block.

The available elements for a definition of a module are:

Option name	Type	Default	Notes
id	string	—	Unique id of the module
description	string	—	Module's description
dependsOn	string's list	—	List of the dependent modules
hidden	boolean	false	Flag to indicate that the module is internal and not shown to the user
excludeReport	boolean	false	Flag to indicate that module will not participate to the final report
input {}	—	—	Rules for the creation of the input file for R code
output {}	—	—	Rules to check the artifacts produced by the R code

What follows is an example of a module descriptor:

```
indicator {
  id = 'module_1' ①

  description = "Description of the statistical module" ②

  dependsOn = ['setup'] ③

  hidden = false ④

  excludeReport = false ⑤

  input { ⑥
    ...
  }

  output { ⑦
    ...
  }
}
```

- ① module's id
- ② module's description
- ③ list of the dependent modules (to be executed before this one)
- ④ hidden flag, if true the module will not be shown in the modules' list
- ⑤ exclude flag, if true the module will not participate to the creation of the final report
- ⑥ input {} block to describe the input of the R code
- ⑦ output {} block to describe the output of the R code

As said before this descriptors defines which data must be prepared for the R code; those data are

in the form of a csv file whose content and name is specified thanks to the `input {}` block.

The available elements into such block are:

Option name	Type	Default	Notes
table	string	—	Name of the master table
fields	string's list	—	List of the fields from the master table
groups	string's list	—	List of the fields from the master table for grouping
criteria	string	—	TBW
sql	string	—	TBW
order	string's list	—	List of the fields from the master table for ordering
file	string	—	Name of the output file

By example, to create a file named `input.csv` like the following:

```
SEX,AGE_RANGE,COUNT
M,1,150
M,2,234
F,1,412
F,2,223
...
```

we can write:

```

indicator {
  ...
  input {
    table = "MAIN" ①

    fields = [ ②
      'SEX', 'AGE_RANGE',
      'count(*) as COUNT'
    ]

    groups = [ ③
      'SEX', 'AGE_RANGE'
    ]

    file = "input.csv" ④
  }
  ...
}

```

- ① master table name
- ② fields to select
- ③ fields for group by
- ④ output filename

What happens with the previous code is:

1. NeuBiro compose an SQL selection against the table named **MAIN** using what specified in the **fields** and **groups** variables
2. Writes the obtained dataset into a file named **input.csv**

Essentially the input block translates to the following SQL statement:

```
SELECT SEX, AGE_RANGE, count(*) AS COUNT FROM MAIN GROUP BY SEX, AGE_RANGE
```

The **input {}** block can also describe multiple files to be created as demonstrated in the following code:

```

indicator {
  ...
  input {
    'FIRST' {
      table = "MAIN"
      fields = [ ... ]
      groups = [ ... ]
      file = "input_one.csv"
    }

    'SECOND' {
      table = "MAIN"
      fields = [ ... ]
      groups = [ ... ]
      file = "input_two.csv"
    }
  }
  ...
}

```

The last element is the `output {}` block, it is used to describe, and to verify, the output from the R code.

In the following snippet we tell to NeuBiro to check for the presence of a file named `report.xml`:

```

indicator {
  ...
  output {
    files = ["report.xml"] ①
  }
  ...
}

```

① list of the files expected at the end of the R code execution

If all the files specified with the `files` variable are not present the indicator will produce an error and the execution will be stopped.

5.2. The select unit specifications

The select units specs permits to modify the behaviour of the statistical packages, feeding the statistical routines with a subset of the data.

```

variables {
  'AGE_RANGE' { ①
    table = "master" ②
    label = "Age range" ③
    type = "string" ④
  }
}

```

- ① field to consider for the extraction of the unique values
- ② table containing the field
- ③ label (currently non implemented)
- ④ type of the field

TBW



This section is draft, work is in progress on the implementation of the select unit.

5.3. The module's statistical code

The statistical code has in charge all the statistical calculation and it's output is represented by an xml file containing a portion of the final report. It receives the input data specified by the module descriptor as described in the previous section and performs all the statistical processing needed by the module producing the expected output and a mandatory file named `report.xml` containing the partial code that will be merged to produce the final report.

What follows is an example for the statistical code:

indicator.r file

```

# =====
# Entry point
#
# baseDir = Root directory for all indicators
# workDir = Work directory for this indicator
#
# =====

data <- read.table(paste(workDir, "/module_1/input.csv", sep=""), header=TRUE, sep=
",", colClasses="character") ①

# do something with the data ②

# write xml file for report ③

rm(list=ls(all=TRUE)) ④

```

- ① load the data to process

- ② do required processing
- ③ write the partial xml code for the final report
- ④ clean up R space

When NeuBiro start to process a module, after the data preparation step, invokes the R interpreter looks for a file named `indicator.r` and pass to the code two variables:

- `baseDir` - the root directory of the statistical package
- `workDir` - the working directory where to write the produced data of this module

Usually a statistical module is quite complex and it is better to decouple the common function for a better code reuse, what follows is an example on how to better organize the code:

```
<ROOT>
|-- modules ①
  |-- commons
  |   |-- tools.r ②
  |   |-- libs2.r ②
  |-- module_1
  |   |-- indicator.specs
  |   |-- indicator.r ③
  |   |-- implementation.r ④
  |-- ...
```

- ① modules root
- ② common files shared by all modules
- ③ entry point for the module code
- ④ implementation of the module code

The R code is divided in multiple files grouping the code in common to exploit the DRY (Don't Repeat Yourself) principle.

In the next snippet we can see an example for an entry point that in turn load the real implementation of the module, the file `implementation.r`, and executes its main function `indicator1()`.

indicator.r file

```
# =====  
# Entry point  
#  
# baseDir = Root directory for all indicators  
# workDir = Work directory for this indicator  
#  
# =====  
  
source(paste(baseDir, "/commons/tools.r", sep="")) ①  
source(paste(baseDir, "/module_1/implementation.r", sep="")) ②  
  
indicator1() ③  
  
rm(list=ls(all=TRUE)) ④
```

- ① load common functions
- ② load module implementation
- ③ executes the implementation code
- ④ clean up R space

The real implementation of the module code can be as follows:

implementation.r file

```

indicator1 <- function() {

  # Set the working directory
  setwd(workDir) ①

  # Load data
  table1 <- read.table(paste(workDir, "/input.csv", sep=""), header=TRUE, sep=",") ②

  # ... do something with the loaded data ...

  # Write report.xml file
  writeTable(file="report.xml", ③
    data=table1,
    append=append,
    vars=c("sex", "n", "perc"),
    headlabs=headlabs,
    headwidth=c("260pt", "60pt", "60pt"),
    colalign=c("left", "right", "right"),
    headalign=c("left", "center", "center"),
    varcolalign=c("align_1", "align_2", "align_3"),
    footlabs=footlabs,
    footnote=footnote,
    title=title,
    section=section,
    graph=NULL)
}

```

- ① set the working directory
- ② load the data from the input.csv prepared by the module descriptor
- ③ write the report.xml file using a custom function

Thanks to the workDir and baseDir path passed to each module it is possible to retrieve all the artifacts produced by all the other modules reusing and minimizing the calculations for complex tasks.

5.3.1. Parameters passed to the R code from NeuBiro

When NeuBiro run the R process to execute the statistical code it send several variables that reflects the choices made by the user using the graphical interface.

The list of the variables is:

Variable	Description
baseDir	TBW
workDir	TBW

Variable	Description
language	TBW
operator	TBW
year	TBW
engine_type	TBW
reference	TBW
reference_files	TBW
input_files	TBW
funnel_group	TBW
select_unit	TBW

Is up to the statistical code to use or ignore them, it is not mandatory, and can be used to implement more complex analysis.

5.3.2. Creation of zip compressed files

Compressed files can be useful to create data packages that can be sent to outside repository or to be used by others. The creation of such a file can be challenging when working in a multi platform environment, in fact R libraries are often different for each platform leading to incompatibility problems.

To overcome these problems NeuBiro provides a specific support to create this kind of file abstracting the underlying operating system.

To create a zip file, the indicator's code, has to simply create the single files that will compose the compressed archive and create a simple descriptor in YAML format; NeuBiro will take care of the creation of the .ZIP file.



The descriptor's filename **must** have the `.zip.yml` extension to be recognized as zip descriptor.

What follows is a sample zip descriptor file:

example.zip.yml

```
# ZIP FILE DESCRIPTOR
file: output.zip ①
files:
  - file_one.csv ②
  - file_two.csv ②
  - file_three.csv|file_three_renamed.csv ③
cleanup: true ④
```

① name of the final zip file

- ② list of files that will compose the final zip archive
- ③ specifies a new name for the file in the zip archive
- ④ cleanup flag, if true NeuBiro will delete the intermediary files leaving only the generate zip file

In the following listing is shown a simple function for the R language to create the specification file.

```
writeZipDescriptorFor <- function(zipfilename, fileslist, cleanup=TRUE) {
  descfilename <- paste(zipfilename, ".yaml", sep="")
  fileConn <- file(descfilename)
  writeLines("# ZIP FILE DESCRIPTOR", fileConn)
  close(fileConn)
  fileConn <- file(descfilename, open="at")
  writeLines(paste('file: ', zipfilename, sep=""), fileConn)
  writeLines(paste('cleanup: ', ifelse(cleanup, 'true', 'false'), sep=""), fileConn)
  writeLines('files:', fileConn)
  for (i in 1:length(fileslist)) {
    writeLines(paste('- ', fileslist[i], sep=""), fileConn)
  }
  close(fileConn)
}
```

The function can be placed in a shared library file (eg `tools.r`) loaded by each indicator's code hence reusable between them.

The following code demonstrate the use of the above described function to create two zip files:

```
1 main <- function() {
2   # Creates some example text files
3   fileslist <- c("descriptor-local.yaml", "one.txt", "two.txt", "three.txt",
4 "descriptor-central.yaml", "four.txt", "five.txt")
5   for (i in 1:length(fileslist)) {
6     createTestFile(fileslist[i])
7   }
8   # Creates a file named test1.zip
9   fileslist <- c("descriptor-local.yaml|descriptor.yaml", "one.txt", "two.txt",
10 "three.txt") ①
11 writeZipDescriptorFor("test1.zip", fileslist, TRUE)
12 # Creates a file named test2.zip
13 fileslist <- c("descriptor-central.yaml|descriptor.yaml", "four.txt", "five.txt",
14 "notpresent.csv", FALSE) ②
15 writeZipDescriptorFor("test2.zip", fileslist)
16 }
17 main()
```

- ① rename the source file

- ② do not delete the composing files

The previous code creates the following descriptors:

test1.zip.yml

```
file: test1.zip
files:
- descriptor-local.yml|descriptor.yml ①
- one.txt
- two.txt
- three.txt
cleanup: true
```

- ① rename descriptor-local.yml into descriptor.yml



Please note how the source files are renamed using the following notation:

```
- original.ext|renamed.ext
```

test2.zip.yml

```
file: test2.zip
files:
- descriptor-central.yml|descriptor.yml ①
- four.txt
- five.txt
- notpresent.csv ②
cleanup: false
```

- ① rename descriptor-central.yml into descriptor.yml
- ② missing file

If a file is missing a log message will be reported to the user and the resulting zip file will not contain created anyway with the remaining files.

The content of the zip files will be as follow:

test1.zip

```
test1.zip
+-- descriptor.yml
+-- one.txt
+-- two.txt
`-- three.txt
```

and

test2.zip

```
test2.zip
+-- descriptor.yml
+-- four.txt
`-- five.txt
```

When NeuBiro, after the execution of the R interpreter, regains control and finds a descriptor file like the one specified above it creates the compressed archive in a cross platform way.

5.4. The report template

The final report is generated using DocBook

```
<ROOT>
+-- ...
`-- report ①
    +-- master.xml ②
    +-- chapter.xml ③
    +-- master.xsl ④
    `-- resources ⑤
        `-- logo.png
```

- ① final report descriptor's root
- ② master file for final report
- ③ master file for each chapter of the final report
- ④ master xsl file for final report formatting
- ⑤ additional resources needed by the report template

6. Example statistical package

NeuBiro installation provides a sample dataset and a sample statistical package to let the user to test the installation out of the box.

The example package can be useful to the reader of this guide to observe a simple working code.

After a successful installation the layout of the installed software will be:

```
C:\NeuBiro-0.7 ①
+-- docs ②
+-- sample-data ③
`-- packages ④
    |-- sample-package ⑤
        +-- import ⑥
        |-- modules ⑦
```

- ① Main installation folder
- ② Documentation in PDF e HTML format
- ③ Sample data set
- ④ Statistical package main folder
- ⑤ Sample statistical package
- ⑥ Import specifications
- ⑦ Statistical modules

The information we are referring to are located under the `sample-package` directory.

6.1. The sample dataset

The standard installation provides a sample dataset to let the testing of the system and to verify that all partes are working.

The sample dataset is composed by two files:

- `sampledata.csv`
- `countries.csv`

6.2. The sample `import.specs`

The sample data described in te previous section can be imported with the import specification provided.

The file `import.specs` provides the code that implements the behaviour described in this guide and shows the techniques for using both the master table and the lookup table.

6.3. The sample statistical package

The sample statistical package defines three modules with the purpose to show various techniques:

Module	Description
setup	Setup module, not shown on the analysis window, performs initializations for the R environment
mod1	Sample module that produces a graph and show the
mod2	Sample module to show how to use data produced by another module
mod3	Sample module to show how to create zip compressed files

A sample report template is provided as well.

7. Hacking NeuBiro

This section describes the organization of the source code and will provide the information needed to work on the code base.

7.1. Accessing the source code



This section should be revised when the code will be publicly available

All the source code for NeuBiro is available on [FIXME](#).

The version control system used is GIT (<http://git-scm.com>) and all the source code can be cloned with:

```
git clone https://to.be.defined/neubiro
```

after cloning the repository, the working copy will have a layout similar to:

```
<ROOT>
+-- src ①
+-- subprojects ②
|   +-- neubiro-app ③
|   +-- neubiro-manual ④
|   `-- neubiro-sample-package ⑤
+-- build.gradle ⑥
+-- settings.gradle ⑦
+-- gradlew.bat ⑧
`-- gradlew ⑧
```

- ① sources for installer
- ② subprojects root
- ③ the java application sources
- ④ the documentation
- ⑤ sample statistical package
- ⑥ main build file
- ⑦ main settings for the gradle build
- ⑧ gradle wrappers for windows and unix

7.2. Building NeuBiro

NeuBiro uses as build tool Gradle (<http://gradle.org>) and thanks to the Gradle wrapper there is no need to install the standalone gradle distribution, it will be sufficient to use gradlew (or gradlew.bat for windows) to run and/or compile the software.

For example, to run NeuBiro in development mode we can issue the following command:

```
./gradlew run
```

The others tasks useful for the project are summarized in the following table:

Task	Description
run	Run NeuBiro in development mode
docs	Builds the documentation in PDF and HTML formats
installer	Creates the installer
dist	Creates all the files composing the distribution (eg software, docs and samples)

The source code for NeuBiro is located at the following path: `<ROOT>/subprojects/newbiro-app`.

7.3. Documentation

The documentation of the project is written using AsciiDoctor (<http://asciidoctor.org>).

The source code for NeuBiro's documentation, divided in users's and programmer's guide, is located at the following path: `<ROOT>/subprojects/newbiro-manual`.

8. References

NeuBiro is build with the help of great open source projects:

Project	Description	URL
The Groovy language	a multi-faceted language for the Java platform	http://groovy-lang.org
The Griffon framework	a desktop application development platform for the JVM	http://griffon-framework.org
H2 Database Engine	embeddable relational database engine	http://www.h2database.com
DocBook	a semantic markup language for technical documentation	http://docbook.org
AsciiDoctor	a fast text processor and publishing toolchain	http://asciidoctor.org

9. Acknowledgements



This section is a placeholder, must be detailed with tasks from each contributor.

All contributions, in any form, should be reported here.

The main developers for NeuBiro are:

Developer	Role
Stefano Gualdi	NeuBiro software, documentation, Java/Groovy programming
Fabrizio Carinci	Statistical analysis and programming of the statistical packages, R Language
Iztok Stotl	Documentation, beta testing