NGLess: A domain specific language for next generation sequence data analysis

Paulo Monteiro - prrm@kdbio.inesc-id.pt

1INESC-ID, Lisbon
2Instituto Superior Técnico, University of Lisbon

November 2014

Abstract
Almost all bioinformatics tasks require interfacing and execution of many programs to process data. Computational analysis is, in many cases, the bottleneck for many scientific facilities as the time required to setup pipelines and to interpret the results reduces productivity. Only a small but growing number of laboratories around the world have the required expertise in both biology and computation.

Presently, most molecular biology laboratories need to deal with huge amounts of sequence data, making the development of more sophisticated computational tools an urgent problem to be solved. In this context, the development of tools with the ability to set up simple analysis pipelines plays an important role in bioinformatics. However, the interfaces between the existent tools are still a problem particularly acute due to the use of non-standards formats. Another important aspect of computational tools is its usability, scalability and robustness. Nonetheless, there are other problems inherent that also require our attention, such as data reproducibility.

To tackle all these problems we propose a domain specific language (DSL), named NGLess, that can be used to specify a series of operations leading to the analysis of next generation sequence data. Unlike Make-like tools, NGLess is able to detect semantic errors, make certain types of errors impossible, and, generally, allow for faster pipeline design.

Keywords: Domain Specific Language, Next Generation Sequence, Bioinformatics

1. Introduction
On a day to day basis, bioinformaticians require interfacing and executing many programs to process data. Computational analysis is, generally, a bottleneck for many laboratories and scientific facilities [7]. In contrast to what one may think, it is not computation time, but the setting up of pipelines and the interpretation of the results that takes most time. This happens because only a small, but growing, number of laboratories around the world have the required expertise in both biology and computation. Other groups may be lucky enough to have a biologically savvy computational expert on hand to delegate such tasks to [17]. So setting up simple analysis pipelines has become an important task that must be possible to be performed by any one that works with biological data.

The task of putting together different programs and tools in order to create a workflow for data analysis is still a time consuming task. This problem is particularly acute when we look at the time that is needed to create parsers, due to the use of non-standards formats.

With the current commercialisation of various affordable desktop sequencers, next generation sequence (NGS) is being made available to more and more biologists. Consequently, NGS data analysis can happen anywhere, e.g in a laboratory or even at home. NGS data analysis has many different applications, and so it is impossible to define a strict pipeline process as it depends on the context.

The complete NGS data analysis process is complex, includes multiple analysis steps, is dependent on a multitude of programs and databases and involves handling large amounts of heterogeneous data. Due to the enormous success of NGS projects, a flood of tools has been created to support specific parts of the analysis workflow. It is clear that the appropriate choice of tools is a non-trivial task, especially for inexperienced users [25].

There are currently many different NGS tools that are commonly used in a normal NGS pipeline, such as:

- **FastQC**:  
- **PRINSEQ**:  
- **NGS QC**:  

1http://www.bioinformatics.babraham.ac.uk/
generate statistics about *FastQ* files;

- Samtools[21], BWA[20], BLAST[3] to align sequences against a genome;
- HTSEQ[4] to annotate a data set;
- IGV[27] to visualise genomic datasets.

To ease the development of new NGS tools a programming language, named Bellman’s GAP [29], was designed.

There are, currently, many different ways to create a pipeline to perform NGS analysis:

- NGS analysis using Linux/Unix tools
- NGS analysis using Workflow Engines systems
- NGS analysis using Libraries
- NGS analysis using Hadoop/MapReduce/HBase frameworks

**Linux/Unix tools**

This is the most popular approach to perform NGS analysis, due to the fact that it requires no setup time, only an UNIX/Linux environment.

In order to perform NGS analysis using UNIX/Linux tools, a researcher must know a few things about the Linux command-line tools, e.g. grep, awk. Also, one is required to have some expertise in some programming language in order to generate results and be familiar with the available NGS tools.

A normal NGS analysis will result in multiple invocations in the Linux terminal. The researcher normally creates a *shell script* or *Makefile* 2 to manage the used invocations and to allow analysis reproducibility.

**Workflow engine systems**

Workflow engines systems (WES) are the most used tools in major research centers that deal with NGS analysis. With the use of interfaces, a workflow of processes is easy to create, but the systems are not easy to setup.

Commonly used WES are Taverna [16], Galaxy[1] and Kepler [2].

**Libraries/Packages**

The use of libraries/packages are another option that users have nowadays to tackle NGS analyses. They can be of easy use if one is already familiar with that language.

The most commonly used tools are Bioconductor [13], Biopython [9] and NGS++ [24].

**Hadoop/MapReduce/HBase frameworks**

Another existent approach available to researchers to tackle NGS analysis is using Hadoop, MapReduce or HBase and typically involves distribution of work across a cluster of machines which access a shared file system, hosted on a storage area network[34]. Message Passing Interface (MPI), and recently Hadoop MapReduce API are used to implement this parallel processing across different machines. Another model used is cloud computing[5]. As stated by Taylor in [34], Cloud computing equals high performance computing + web interface + ability to rapidly scale up and down for on-demand use.

There are some implementations that allow to perform NGS data analysis as Hadoop-Bam [23], SeqPig [32], Cloudburst [30] and Myrna [19].

It is very important in bioinformatics, as in other areas, to have applications that have good usability, scalability and robustness. But there are other problems inherent that also require our attention, such as data reproducibility [28], either by the researcher or by some third-party, enabling the verification of data.

Nearly all NGS applications rely on sequence alignment as the first analysis step [23]. Before the alignment researchers require some kind of preprocessing of data, that is always dependent on the researcher interest. Figure 1, represents a generic NGS analysis pipeline, based on a survey created by Pabinger et al.[25] for variant analysis.

The main goal of this work is to create a tool that enables the creation of a pipeline of work for all first phase of NGS data analysis until the point (inclusive) of annotation. We want to do this while achieving the following objectives:

- Improve the development of NGS data analysis tools;
- Facilitate the configuration and execution of pipelines;
- Enable data analysis reproducibility;
- Maximise the computational resources available;
- Process a dataset of any size.

We also want to provide the user with a easy to use web application that enables the result visualisation and the creation of scripts using an editor or a wizard.

The proposed tool is a domain specific language (DSL), named *NGLess*, where the user can specify a series of operations leading to the desired result.
Unlike Make-like tools, NGLess is able to detect semantic errors, make certain types of errors impossible, and generally allows for faster pipeline design than is currently possible.

2. Domain Specific Languages

In engineering, approaches that are generic can be distinguished from those that are specific. A general approach provides a generic solution for many problems but it might not be optimal. As for specific approaches, they provide a much better solution for a smaller set of problems.

Domain-specific languages have been in use for decades. In Unix there is a tradition of such small languages [6] including sh and bash (for shell scripting), lex and yacc (for lexical analysis and parsing), and make (for automated software builds). These are external DSL systems, that typically use Unix built in tools to help with translation.

Older programming languages as Cobol, Fortran and Lisp appeared as domain languages to solve problems in certain areas (business processing, numeric computation and symbolic processing). These programming languages have gradually grown into general purpose solutions and the need, for support, to solve specialised domains re-emerged.

A DSL offers possibilities for analysis, verification, optimisation, parallelization, and transformation in terms of DSL constructs, that would be much harder or unfeasible if a general purpose language (GPL) had been used. Most of the time the GPL source code patterns involved are too complex or not well defined [22].

A domain specific language (DSL) is defined by Deursen, Klint, and Visser[11] as:

**Definition 2.1** (Domain Specific Language). Is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain

In a shorter definition, a domain specific language is a computer programming language of limited expressiveness and specific to a given domain.

2.1. Implementation

Different approaches can be taken to create a DSL, depending on the purpose of the tool. As Fowler defines, domain specific languages can be divided into three main categories[12]: External DSL, Internal DSL and Language Workbench.

It is important to mention that there is not a best approach. One of the possible options may apply better for different problems, even at the same domain context. It is not easy to choose the best path to follow, but should be highly thought through as it completely changes the DSL structure. In this work an external approach was used.

2.2. Trade-offs

There are many advantages and disadvantages when adopting a DSL approach to a given domain. It is important to consider if the opportunities outweighs the risks for a given context. The advantages of domain specific languages has been described by [Mernik, Heering, and Sloane, Deursen, Klint, and Visser, Spinellis] at [22, 11, 33] as:

- Development productivity
- Concise, domain specific notations
- Analysis and verification
- Platform independence
- Abstraction
- Domain-specific tooling support

There are also disadvantages when using a DSL. One of the biggest disadvantages of a DSL is what Fowler[12] calls as the cacophony problem. It is the concern that languages are hard to learn, so using many languages will be much more complicated than using a single one. Multiple languages makes it harder for new people to join new projects as the learning curve may not pay off the limited scope of the DSL.

A key aspect, in domain specific languages, is the expressiveness of the code, as it allows a bioinformatician to improve his productivity by developing pipelines in a, much, simpler way. Many of the pipelines created for NGS data analysis may take many days to complete and an error
may occur a long time after the beginning of the pipeline resulting in many hours lost. When using a DSL it is possible to have a much richer type validator, oriented to NGS data analysis, that identifies errors before the interpretation.

By using a DSL it is possible to easily reproduce an analysis by simply sharing a DSL script. This allows the scientific community to share the pipelines (e.g., scientific papers) that lead to their results.

We believe that the advantages of using a DSL for NGS data analysis outweigh the disadvantages.

3. Next Generation Sequence

Between 1990 and the publication of a working draft in 2001, more than 200 scientists joined forces in a 3 billion dollar [15] effort to read the roughly 3 billion bases of DNA that comprise the human genome [18]. The cost was too high and it was of key importance to develop a new method of sequencing that was both cheaper and faster.

This demand has lead to the development of next generation sequence (NGS) platforms. In the past decade, several NGS platforms have been developed to provide low-cost and high-throughput sequencing [15]. There are three main companies that built machines that allow NGS data analysis:

- Illumina
- Applied Biosystems
- Roche

In the past few years genome sequencing technologies have exceeded Moore’s Law and the speed of genome sequencing has more than doubled every two years since 2003 due to NGS platforms, see Figure 2 [8].

An example of a FastQ file with a single read looks like the following (abbreviated in the sequence and quality field):

```
@SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGCTTTTTTGTTTGGAACCGAAAGG
+SRR014849.1 EIXKN4201CFU84 length=93
3+&$#""""""""""""""""""""""""""""
```

Figure 2: Evolution of sequencing cost

3.1. Input Data

All companies follow the standard defined for FastQ [10], where each read follows the format:

```
@title and optional description
sequence line(s)
```

+optional repeat of title line quality line(s)

Typically, a FastQ file has from 1 to 10GB compressed, and each step in the analysis generates more reads, resulting on a huge amount of Disk/Ram required. The meaning of each line of the read and the possible values it can contain are:

1. **Title line** often holds just an identifier. This is a free format field with no length limit.
2. **Sequence line** is a string of text written in an alphabet of 4 characters and representing a biological sequence of DNA. The characters in this type of sequence are named base pair (BP), and can be one of the four bases Adenine (A), Guanine (G), Cytosine (C), and Thymine (T), sometimes also called nucleotides.
3. **Delimiter** is used the character + to signal the end of the sequence line and the start of the quality string. Originally this also included a full repeat of the title line text.
4. **Quality line** encodes the quality values for the sequence. Different encoding might be used, see Table 1.

The quality string must be equal in length to the sequence string.

<table>
<thead>
<tr>
<th>Description</th>
<th>ASCII</th>
<th>Quality score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sanger standard</td>
<td>33 – 126</td>
<td>0 to 93</td>
</tr>
<tr>
<td>Solexa/early Illumina</td>
<td>59 – 126</td>
<td>-5 to 62</td>
</tr>
<tr>
<td>Illumina 1.3+</td>
<td>64 – 126</td>
<td>0 to 62</td>
</tr>
</tbody>
</table>

Table 1: The range values for the three possible FastQ variants.

An example of a FastQ file with a single read looks like the following (abbreviated in the sequence and quality field):

```
@SRR014849.1 EIXKN4201CFU84 length=93
GGGGGGGGGGGGGGCTTTTTTGTTTGGAACCGAAAGG
+SRR014849.1 EIXKN4201CFU84 length=93
3+&$#""""""""""""""""""""""""""""
```

Quality uncertainty

There are two different equations, as Cock et al. in [10] mentions, that can be used to calculate the quality of each nucleotide. The difference in both equations is the encoding of the odds $\frac{p}{1-p}$ versus $p$ (where $p$ is the estimated error probability for a base pair), but they generate similar results differing only at low quality values.
Also, different sequencers may encode the quality sequence using different, ASCII, range values. These possible ranges can be consulted in table 1. It is important to know the exact encoding used to generate the FastQ file, as different encodings lead to different results.

### 3.2. NGS Operations

Before proceeding to the currently existent tools, it is important to explain the most commonly used operations in NGS data analysis, see Figure 1.

#### Quality Control

The quality of the data is affected by several factors regardless of the NGS platform. Quality control (QC) is an essential step to ensure that the data used for analysis is not replete with low-quality sequences that can lead to incorrect conclusions [31]. The most common way of performing QC is by looking at summary statistics of the data. There are different programs that can produce those statistics and the results can be seen through a web application.

#### Pre-process

Pre-processing, is a key operation known in many other areas as “garbage in, garbage out”, where the idea is to filter everything that could possibly generate bad results. Normally, pre-process, is applied after doing QC. A different pre-processing approach can be applied to the same data and is completely dependent on what the user finds of key importance.

There are two key operations that are the most commonly used when performing pre-processing.

- **Remove Duplicates**: Sequence replication can occur during different steps of the sequencing protocol and can therefore generate artificial duplicates [14]. A common step is to remove duplicate occurrences of a given read, or to keep at most N copies.

- **Trim**: There are two types of trimming, static and based on quality.
  
  Static trimming is the process of reducing sequences to a specific length by removing, at least one of, the edges of a read. The edges correspond to the bases from the 5’ and 3’ end.

  Trimming based on quality is the process of reducing the quality sequence to the biggest subsequence with a minimum quality.

#### Alignment

When two DNA sequences are arranged so that their most similar base pairs (BP) are overlapped, they are said to be aligned. The overlaps do not have to be exact, and for each BP in a alignment it can either be a match (if both BP are equal), a mismatch (if BP are different) or a gap (if BP from one sequence is aligned with spaces, representing the insertion or deletions of portions of DNA from one of the sequencers).

Many bioinformatics tasks depend upon successful alignments. The SAM (Sequence Alignment/Map) file format which can be optimised to BAM (Binary Alignment/Map), is the standard defined to describe sequence alignments. All alignment algorithms use this format as the output format.

#### Annotation

An annotation, independently of the context, is the process of adding a note by way of explanation or commentary. Genome annotation is the process of identifying the locations of genes and all of the coding regions in a genome and determining what those genes do. Once the sequences are aligned (Section 3.2), annotation can be performed using a General Feature Format (GFF) file.

There are many publicly accessible databases that contain those GFF files as Ensembl\(^3\), NCBI\(^4\) and UCSC\(^5\).

### 4. NGLess description

Our solution passes by the creation of a Domain Specific Language (DSL)[12] for NGS analysis and also the design of a graphical user interface that allows users to see statistics about the datasets.

With this DSL we want to provide a researcher with a context aware tool that allows for the whole first phase of NGS analysis, without having to worry with any of the previous mentioned problems. This first phase mentioned includes:

2. Pre-processing of a data set;
3. Mapping against genomes;
4. Annotation of the data set;
5. Visualize the data;
6. Keep track of every run made with NGLess.

#### 4.1. NGLess script example

An example of our language that would generate statistics to multiple files, remove not unique sequences, pre-process the dataset with a block of operations in each read, that would map and annotate and finally write to a file the results (in this

\(^3\)http://www.ensembl.org/index.html
\(^5\)http://www.genome.ucsc.edu
case counts of annotated genes) can be seen at Listing 1.

```plaintext
ngless "0.0"
/*
   File paths
*/
fp = ['ctrl1.fq', 'ctrl2.fq']
input = fastq(fp)
input = unique(input, max_copies=2)
preprocess(input) using |read|
   read = read[5:]
   read = substrim(read, min_quality=26)
   if len(read) < 31:
      discard
// Human reference
human = "hg19"
m = map(input, reference=human)
f = [{gene}, {cds}]
a = annotate(m, mode="union", features=f)
c = count(a, counts=[{gene}])
write(c, ofile="counts.csv", format={csv})
```

Listing 1: NGLess script example

4.1.1 Highlights

This section has the objective of explaining the key advantages of our Domain Specific Language based on the previous example and in here we intend only to provide some highlights that we believe to be important mention.

Version declaration

The first line of the example, has a version declaration and it must be present in any NGLess script. The version declaration is very important, since it allows for a researcher to reproduce analysis[28], following a given version of our DSL. Any researcher can reproduce a NGLess script created by someone else, since this version not only keeps the version of the script but also knows the versions of the other programs used by the DSL, achieving in this manner full analysis reproducibility.

FastQ

FastQ files can have multiple different quality encodings [10] and so it is of key importance to automatically discover which is the encoding on the file. To achieve that, we do a quality prediction based on the lowest character present for all the quality sequences. Quality control is mandatory and cannot be skipped and, as such, this function generates statistics about the datasets used to allow a researcher to perform a better pre-processing. These statistics are:

- Number of sequences
- Percentage of guanine and cytosine (% GC)
- Minimum and maximum sequence length
- Mean, median, 25th percentile and 75th percentile for each base pair

These statistics are then written to a JSON file and can subsequently be viewed by our visualisation tool.

Pre-process

Another highlight in the example is the `preprocess` function that filters a given dataset. In the example many different operators can be used as conditions (if), indexation (i.e var[10:], var[:], etc.), length (operator len) and many different unary and binary operators.

It is also provided functions that are commonly used in the biological domain as `substrim` that performs a trimming based on a minimum quality [10]. At the end of the `pre-process` is performed quality control (QC) on the resulting data set.

Map

The argument `reference` can either be a path to a data set or the name of a provided reference genome by NGLess. In case the used reference genome, in this case "hg19", is not available locally it is automatically installed.

Annotate

Our implementation requirements, for annotation, are very similar to the ones provided by HTSEQ-count [4]. The main difference is that we add a few more parametrised options to allow/deny ambiguity and also we allow to filter by more than one genomic feature at a time.
Type Validation

It is crucial to detect and report an error as soon as possible, and in a concise manner. The fact that the domain is so restrict allows us to type validate everything in a very strict manner.

It is important to mention that it is not required the direct use of a value in a function for type validation to occur. For example, if one of the file paths ‘ctrl1.fq’ or ‘ctrl2.fq’, associated with variable fp, do not exist an error is reported.

Auto-Comprehension

It is essential to notice that the function fastq receives a list, being this a characteristic of our language that any function of type $\lambda \rightarrow * \rightarrow B$ can be automatically used as $[A] \rightarrow * \rightarrow [B]$.

4.2. External Dependencies

NGLess can be downloaded in two different ways: **source** or by a **binary distribution**.

To build from source we provide a Makefile that in a first stage deals with the build of NGLess and in a second stage with the installation. The installation destination (default /usr/local/) can be chosen by the user.

The binary distribution is a self-contained tarball that contains a NGLess executable and all dependencies.

As such, regardless of the way NGLess is downloaded it can be executed from anywhere. To make sure that all external dependencies (e.g., html libraries) are always accessible by NGLess executable, we only use **relative paths**. The structure of a NGLess installation at directory dir is as follows:

```
   dir/
   |__ bin/
   |     |__ ngless
   |__ share/
   |     |__ ngless/
   |       |__ genomes/
   |           |__ bwa/
   |               |__ html/
```

The directory bin/ is where we keep the actual NGLess executable. The directory share/ is where we keep all the dependency data. Independently from where the user is executing the DSL, NGLess knows at all times where the dependencies are. For example, **bwa** is accessible by doing ../share/bwa/ from NGLess executable location.

4.3. Visualisation

NGLess keeps metadata for the most important results and information, so that the results can be both reproducible and visualised in the future.

We don’t know a priori the structure of the HTML page, and it has to be generated dynamically. Besides using JavaScript and HTML, we also use **AngularJS**, that is a structural framework for dynamic web applications.

The web design is responsive (RWD). This means that the web page will adapt across a wide range of devices (from mobile phones to desktop computer monitors).

Using NGLess, a local web server can be launched at a specific port, in figure 3 port 8000, and all runs are visualisable.

4.3.1 Interpreter results to Browser

We want the user to be able to analyse his results anytime after the execution. In order to do so, for a given script a directory with the script name concatenated with .ngless_outputs is created. All dependencies are installed in that directory. All future executions of that script will also write the results to that directory. Figure 4 shows the flow of the script from execution until visualisation.

4.3.2 Quality Control

Quality control can be present at two different times for a given script. One is mandatory and happens
before any pre-process, the other is optional and occurs after the pre-process.

Figure 5 shows the quality control (QC) statistics, before pre-process, for an example sample with only the mean visible. The corresponding statistics are the aforementioned ones at Section 4.1.1.

Figure 5: Statistics for a sample file

4.3.3 Counts
This mode of visualisation shows the results of the annotation. It is possible to sort (ascending or descending) by the amount of counts, and in this simple manner create a top 10 of the genomic feature with most counts. It is also possible to filter a genomic feature by the gene name. Illustrated in Figure 6, an example with descending applied to column counts.

Figure 6: Top 10 genes for file 'CountResults'

4.4. Create Scripts
NGLess provides two modes to create scripts, see Figure 7. One targets skilled users as every single character of the code has to be written and the other targets users with few programming skills. The solutions are an editor and a wizard. The wizard is an asset for researchers that are starting to learn NGLess as it is a great starting point.

The wizard, developed in AngularJS, enables the creation of scripts while responding to simple questions related to NGS data analysis. This approach eases the script development process as it is only required to deal with natural language to create a NGLess script. The wizard structure is, continuously, updating dynamically, and therefore can be seen the currently developed script at any time.

Figure 7: Wizard and Text Editor

5. Experimental Results
Due to the limitation in size, we will provide an overview on NGLess execution as a whole instead of each feature independently.

To generate the results we pre-processed a dataset, removed duplicates, mapped it against the human genome (hg19) and counted the reads that overlap with known genes.

5.1. Data Context
These performance results were obtained considering a real dataset stored at EMBL-EBI. The data can be accessed at http://www.ebi.ac.uk/ena/data/view/SRP023199 and represent HeLa cells. The FastQ original file has 5.43GB and was reduced to a 5GB sample.

5.2. Discussion
All results related with time were obtained of using a benchmark that samples each test 100 times. The results are presented with an upper and lower bound of 95% confidence.

<table>
<thead>
<tr>
<th>Function</th>
<th>Low Bound</th>
<th>Mean</th>
<th>Up Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastq</td>
<td>54.4 s</td>
<td>54.9 s</td>
<td>55.6 s</td>
</tr>
<tr>
<td>unique</td>
<td>794.2 s</td>
<td>797.5 s</td>
<td>803.1 s</td>
</tr>
<tr>
<td>preprocess</td>
<td>988.6 s</td>
<td>1016.0 s</td>
<td>1045.6 s</td>
</tr>
<tr>
<td>map</td>
<td>5459.0 s</td>
<td>5964.0 s</td>
<td>6205.0 s</td>
</tr>
<tr>
<td>annotation</td>
<td>487.9 s</td>
<td>489.8 s</td>
<td>491.8 s</td>
</tr>
</tbody>
</table>

Table 2: Mean of the NGLess functions execution time

From Table 2, we can see that the map function is the operation that takes the most time to execute. Figure 8 shows that for a sample with 5GB, 72% of the total execution time takes place at the map operation. This percentage can even increase if it is required to calculate the index for the alignment. As for all the other operations, the sum is only 28%
of the overall execution time.

It is important to mention that the map function can execute in parallel and if we allow ngless to run with 8 threads, the map execution percentage reduces from 72 to 28, see Figure 9. Figure 8 and 9 have total execution times of 8321 and 3284 seconds, respectively. This allows to infer that with 8 threads, there is a speedup of approximately 2.5.

5.2.1 Time comparison with similar tools

The features provided by the annotation function are very similar to the ones provided by the HTSEQ-count. As such, we can compare the times for the same SAM sample.

For a common parametrisation, using mode union, NGLess and HTSEQ-count took 490 and 2383 seconds to execute, respectively. We conclude that NGLess produces the same results being 4.8 times faster.

Acknowledgements

This work is supported by national funds through FCT, under project PEST (PEstOE/EEI/LA0021/2013) and DATASTORM (EXCL/EEI-ESS/0257/2012).

References


