Multiscale Complex Genomics

Project Acronym: MuG
Project title: Multi-Scale Complex Genomics (MuG)
Call: H2020-EINFRA-2015-1
Topic: EINFRA-9-2015
Project Number: 676556
Project Coordinator: Institute for Research in Biomedicine (IRB Barcelona)
Project start date: 1/11/2015
Duration: 36 months

Deliverable 6.1: Design of computational architecture of software blocks

Lead beneficiary: University of Nottingham
Dissemination level: PUBLIC

Due date: 31/10/2016
Actual submission date: 31/10/2016

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676556.

Document History

Version	Contributor(s)	Partner	Date	Comments
0.1	Josep Ll. Gelpi Rosa Badia Pau Andrio	BSC	15th Jul 2016	First draft
0.3	Marco Pasi Charles Laughton	UNOT	23rd Aug 2016	Version 0.3
0.4	Marco Pasi	UNOT	26/10/2016	Final details
1.0	-	-	31/10/2016	Final version approved by technical and supervisory boards.

Table of Contents

1 EXECUTIVE SUMMARY

The MuG VRE aims to provide an integrated software environment for researchers in the field of multiscale complex genomics, by facilitating the access to large-scale heterogeneous data and to high-performance software tools, ultimately to rationalise and speed up the scientific process towards understanding the 3D structure of the genome. This document sets the requirements and specifications for the software architecture of the tools section of the VRE, in particular taking into account the heterogeneous needs of the wide potential user community and the technical necessities for data integration and software interoperability and sustainability.

2 BACKGROUND

The Multiscale Complex Genomics (MuG) VRE aims to provide an integrated software environment to fulfil the needs of the research community around the 3D structure of the genome. This is an enormous task as types of data involved range from sequence and annotation data, through either atomistic or coarse-grained simulation data, to high level experimental data like HiC, or FISH. Defining a Data model itself is the initial challenge (see D3.1, and D4.1). A data model should allow to connect the various levels of complexity involved and allow to browse them using a single coordinate system. Tools are the second, and complementary challenge. Although the amount of analysis software built on the different levels of genomics studies is large, there is no previous attempt to build a fully interoperable software suite that supports the integration of the data levels considered in MuG. Interoperable platforms are however not new to genomics. In fact, there is already a general agreement in the community about the need of discovering and accessing data and tools, using a unified and standardized environment [1], and there exist already a number of initiatives to provide and support access to bioinformatics tools [2-7]. A fully interoperable scenario requires a number of components: Common data ontologies providing both machine-readable data type definitions and semantics; service registries allowing easy discovery; and a process management software. In the present scenario, and through the impulse of the ELIXIR initiative [8], projects like EDAM ontology [9], a product of the effort made in the EMBRACE project [10], BioXSD [11], or the Elixir tools catalogue [12] seek to configure a fully interoperable environment for Bioinformatics. However, although this initiative is expected to succeed, it is still far from being a reality. Fortunately MuG VRE does not require to define a universally accepted standard, however it does need to fulfil the requirements of such interoperable scenario within MuG software offer. In any case, the design will align with general interoperability recommendations promoted in the Elixir infrastructure.

This document will describe a proposal of software architecture designed in such direction, taking into account that MuG data model is still under development (WP3 and WP4). The proposal will be also aligned with the implementation prototype provided in D5.1 (WP5)

3 SOFTWARE REQUIREMENTS

The main requirement of MuG software stack will be to provide a software architecture to efficiently use the analysis modules and tools generated by WP6, consuming and producing data, according to the models designed in MuG WP4. The summary of the requirements of such software architecture follows:

1. **Modular architecture.** Tools should be orchestrated as modules of a well-defined functionality with a fully defined input and output data specification. This will allow to encapsulate analysis workflows in a flexible and reusable manner, with minimal programming overhead.

2. **Interoperable modules.** A modular approach cannot be useful unless input and output data are fully compatible, in a way that modules can be freely interconnected in larger workflows without recoding. Full interoperability will require to build a complete data description ontology. The software architecture should take into account eventual changes or evolution of data models, allowing adaptation with minimal programming work. Data interchange through disk and memory should be available.

3. **Module functionalities.** MuG will cover a large number of already existing bioinformatics functionalities that have not been necessarily conceived to interoperate with each other. Additionally, it is not realistic to think of recoding existing software to address specific MuG requirements. Software modules should, then, allow to use existing software, run in its most efficient environment, and allow eventually the use of existing parallelization solutions.

4. **Grouped operations.** A modular design is normally implemented building small modules with well-defined functions. However, some workflows representing more complex operations are usually run as single blocks. The software design should allow to prepack complex workflows as a second level of reusable building blocks. This will allow to optimize internally such workflows in the most appropriate manner.

5. **Module and workflow configuration.** Most software modules will encapsulate existing tools with a large number of configuration options. To appropriately setup such tools, a uniform, both for modules and workflows, configuration mechanism should be provided. Default options should be provided to allow a reasonable use for non-experts.

6. **Workflow management.** The architecture of software modules should be compatible with the chosen workflow management system, and allow to execute the workflows in a number of computer environments (desktop for testing purposes, virtualized environments, and HPC).

7. **Data Access.** In the genomics and structural fields, access to data public repositories is key. There is a growing agreement on the use of RESTful services to provide data. Software design should include the necessary modules to obtain data from the MuG relevant repositories (IRB and EMBL-EBI) through the use of such technology.

8. **Installation & Documentation.** MuG software suite will be the engine supporting the back-end of the central MuG browser and data repository. However, it is not unreasonable that the complete suite or parts of it (given its modular nature) should be installed in other places. This will allow to extend the functionality to, for instance, private or sensitive data repositories. In consequence, a reasonable installation, and verification protocol should be provided. Also, installation and use of the software should be fully documented, and testing data sets available.

4 ARCHITECTURE PROPOSAL

The indicated requirements can be fulfilled with a number of possible solutions, with different programming languages and data supports. Most of these solutions are equally efficient, and choosing among them is usually matter of personal preferences. The proposal of software architecture for the MuG project is based in a number of pre-existing conditions relating both to the perceived needs of the VRE user community and to the partners' expertise and tradition. In particular, the following has been taken into account:

1. Software architecture of the underlying tools is varied (programming languages from FORTRAN to C++, specialized languages like R, and scripting languages including Perl, Python, and others). As a general case, it has been considered that the underlying tool will be called through a single command line with appropriate options, either in the command

line or as a configuration file. Other scenarios are also envisaged (see next section), to facilitate the integration of a wide range of tools in order to meet the needs of the user community.

2. Raw data will be available either as disk files, in the native format generated by data providers (ex. PDB or FASTA formats), or obtained from a RESTful service (IRB or EMBL-EBI).

3. The preferred task manager will be BSC's COMPSs programming model (in particular its Python binding PyCOMPSs).

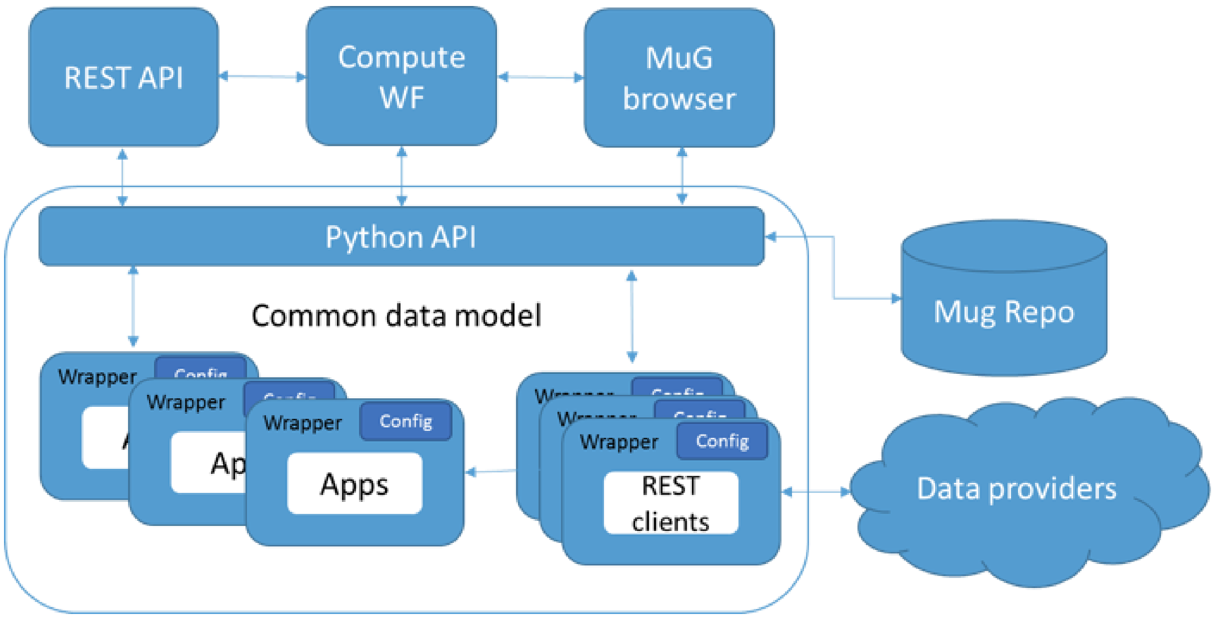


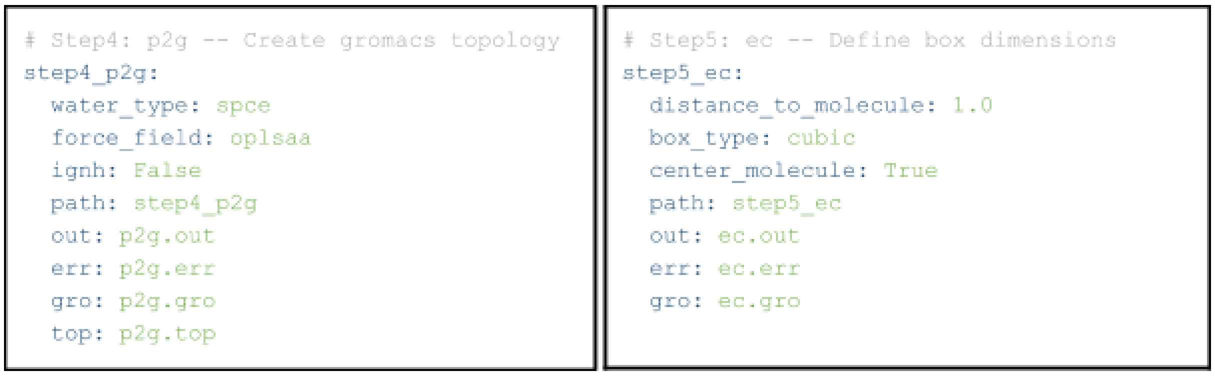
Figure 1. Global schema of the software architecture

4.1 Software blocks

The basis of the MuG software infrastructure would be a series of building blocks organized as a library of modules, encapsulating the necessary functionalities. Following the above considerations, modules will be generated as configurable Python objects wrapping the original software. Interaction with the underlying software will be through command line execution, or, when appropriate, through a specific Python API provided by the software. This ensures that the original software can be kept untouched, minimizing installation and configuration issues. Besides, parallelization strategies already available in such applications can also be used when appropriate. In general terms, wrappers will expose tasks and their dependencies, such that the underlying computational infrastructure can optimise their execution. Our task-based strategy for parallelism is commonly used in a number of runtime environments for high-performance computing (for example, see the RADICAL Pilot project [8], the Extasy project [9], and the recently developed Intel® Threading Building Blocks [10]).

Configuration of the modules will be made through YAML/JSON scripts (see an example in Figure 2). Wrappers will take care of interpreting configuration scripts and translate the setting to the execution command line. This configuration strategy will allow to maintain a stable interface even in the eventual modification of the internal applications, and will hide this complexity to the users. Default configuration schemas will be provided, in a way that non-expert users could execute software with a set of recommended settings.

The functionality of software blocks would be kept to single operations, typically with a minimum set of input data items, and a single output data item, thus maximizing the modularity, flexibility and interoperability. However, it can be foreseen that some sets of operations will be usually performed as a block. In those cases, it is reasonable to build higher level blocks, including a more complex pipeline, made itself from the combination of simpler blocks. These pipelines will be organized in the same way and will offer a similar interface and configuration strategies than simpler blocks.



```
# Step4: p2g -- Create gromacs topology
step4_p2g:
  water_type: spce
  force_field: oplsaa
  ignh: False
  path: step4_p2g
  out: p2g.out
  err: p2g.err
  gro: p2g.gro
  top: p2g.top
```

```
# Step5: ec -- Define box dimensions
step5_ec:
  distance_to_molecule: 1.0
  box_type: cubic
  center_molecule: True
  path: step5_ec
  out: ec.out
  err: ec.err
  gro: ec.gro
```

Figure 2. Examples of configuration files.

An important issue when combining different applications is the compatibility of their software environments (preferred operative systems, system libraries, etc.). It is not reasonable to think that a relatively complete software library as planned for MuG, will be possible in a single software environment. Virtualization is the usual strategy to address these issues. Depending on the specific requirements, we will use Docker containers to encapsulate operations that require a complex system setup. Dockerized applications can be easily run through a simple command line, and even though they can encapsulate complex systems, they will still be compatible with the wrappers. The complete library or part of it would be also encapsulated in either virtual machines or Docker containers for distribution or execution in other environments.

Based on these considerations, software blocks that implement Tools within the VRE are defined by the following four parameters or features:

1. **Functionality:** The specific functionality that the tool provides, should match the envisaged use cases within the field of multiscale complex genomics.
2. **Required Input Data:** The type (or types) of data that the tool requires as input in order to perform its function; these should be only valid data types, as defined in the MuG Data Management Plan (see D4.2 and Section 4.3).
3. **Resulting Output Data:** The type (or types) of data that the tool produces as a result of its execution; the same limitations apply here as for inputs, ensuring that tools can be easily combined into workflows.

4. **Type of integration within the VRE:** Details of how the tool is integrated within the VRE, including specific information about its execution environment requirements and dependencies.

Workflows (i.e. sets of Tools to be run in a specific order) will also be defined in a similar way. It is important to note that abstracting the definition of a tool to these four simple features is a significant step towards making the software architecture of the VRE robust to future changes, thereby improving its long-term sustainability.

4.2 External Data Access

Operations required in MuG will make extensive use of data repositories, either public ones like Ensembl, ENA, EGA, PDB, Uniprot, BIGNASim or those generated within the MuG project. Whereas the internal structure of such repositories is diverse, there is a general agreement in using RESTful interfaces to access data, and most of them provide or will provide such interfaces. MuG software library will include specific modules to access such services, and eventually adapt the data to the internal data representation. Additionally, a RESTful interface to access MuG internal data will be generated following the appropriate recommendations (See DMP D4.1 and D4.2).

4.3 Data Interoperability

The whole design of a modular software library requires to ensure the complete interoperability of the modules. The use of common types of interface is the first step. MuG software modules will be used through Python scripting and will receive input and output data using the Python object schema, either with in-memory objects or serialized for disk based storage. The use of other data formats will be restricted to the internal applications and hidden from the external usage of the modules. A more relevant issue is the design of a common data type schema covering all data used by software modules. This does not mean that a single data model for all levels (see D3.1) of MuG data should be established. Instead, the most appropriate data model for each level will be chosen. The integration of such models will ensure the interoperability: compatibility within the description of molecular entities from atomistic to coarse-grained levels, common genomic coordinates, etc.

4.4 Workflow Management

As indicated above, workflow management will be based in PyCOMPSs, the Python binding of COMPSs programming model. COMPSs allows to exploit implicit parallelism in task-oriented workflows at run-time, and is able to control virtualized systems following the need of the workflow in a dynamic manner. COMPSs provides a run-time environment for clusters, large HPC systems, and also grid and cloud systems, including clusters managed by Docker technology. PyCOMPSs syntax is based on the use of decorators that indicate to the COMPSs run-time which methods will become tasks (nodes in the workflow). The use of Python modules will allow to define any complex workflow just as a Python script that will call MuG modules as internal tasks. Workflows could be configured using the same YAML/JSON procedure as modules, in a way that a single file will manage the configuration of the individual modules and the complete workflow.

4.5 Verification and Benchmarking

Integration of external software in the VRE, especially in cases where complex installation processes are involved, will require a final step to assess that the software is executing correctly within the MuG computational infrastructure. This step should ensure that the software *(i)* is producing the correct output, *(ii)* with the expected performance: both aspects are of fundamental importance in order for users to confidently and efficiently use the VRE. We envisage two scenarios in order to achieve this, which are described in the following.

It is widely considered as good software development practice to include in a software package a *test suite* to automatically verify correct execution, which administrators are advised to run in order to test the installation: this will be the preferred verification strategy for those packages where such a feature is available. Alternatively, developers often provide example outputs of their software packages, either bundled with the software or available on a web page. MuG administrators will evaluate correct execution comparing the VRE outputs to these standard outputs in cases where these are available, and possibly negotiate with external software developers to make standard outputs available. Even in these cases, VRE users may wish to further assess themselves the output of tools; in fact, during an initial requirement survey carried out within the MuG project [11], it was pointed out by several prospective users that the VRE should include multiple tools with similar or identical functionality, in order to leave the freedom of choice to the user. The presence of multiple equivalent tools within the VRE would allow MuG administrators to assess correct execution by comparison, in those cases where this is applicable.

5 REFERENCES

1. Stein L. Creating a bioinformatics nation. Nature 2002;417(6885):119–120.
2. Bhagat J, Tanoh F, Nzuobontane E, Laurent T, Orlowski J, Roos M, et al. BioCatalogue: A universal catalogue of web services for the life sciences. Nucleic Acids Res 2010; 38(Web Server):W689-W694.
3. Goble CA, Bhagat J, Aleksejevs S, Cruickshank D, Michaelides D, Newman D, et al. myExperiment: a repository and social network for the sharing of bioinformatics workflows. Nucleic Acids Res. 2010; 38 (Web Server):W677–W682.
4. Hull D, Wolstencroft K, Stevens R, Goble C, Pocock M, Li P, Oinn T Taverna: a tool for building and running workflows of services. Nucleic Acids Res 2006; 34:729-732.
5. Goecks J, Nekrutenko A, Taylor J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol. 2010;11:R86.
6. BioMoby Consortium, Wilkinson MD, Senger M, Kawas E, Bruskiewich R, Gouzy J, et al. Interoperability with Moby 1.0 - It's better than sharing your toothbrush! Brief. Bioinform. 2008; 9(3):220-231.
7. Repchevsky D, Gelpi JL. BioSWR – Semantic Web services Registry for Bioinformatics PLoS ONE 2014; 9(9): e107889.

8. http://radical-cybertools.github.io/radical-pilot

9. http://www.extasy-project.org/

10. https://www.threadingbuildingblocks.org/

11. Report for the internal consulation IC6.0, available on the MuG website.