ARTENOLIS: Automated Reproducibility and Testing Environment for Licensed Software

Laurent Heirendt & Sylvain Arreckx, Christophe Trefois, Yohan Yarosz, Maharshi Vyas, Venkata P. Satagopam, Reinhard Schneider, Ines Thiele, Ronan M.T. Fleming

Abstract

Motivation: Automatically testing changes to code is an essential feature of continuous integration. For open-source code, without licensed dependencies, a variety of continuous integration services exist. The CONstraint-Based Reconstruction and Analysis (COBRA) Toolbox is a suite of open-source code for computational modelling with dependencies on licensed software. A novel automated framework of continuous integration in a semi-licensed environment is required for the development of the COBRA Toolbox and related tools of the COBRA community.

Results: ARTENOLIS is a general-purpose infrastructure software application that implements continuous integration for open-source software with licensed dependencies. It uses a master-slave framework, tests code on multiple operating systems, and multiple versions of licensed software dependencies. ARTENOLIS ensures the stability, integrity, and cross-platform compatibility of code in the COBRA Toolbox and related tools.

Availability and Implementation: The continuous integration server, core of the reproducibility and testing infrastructure, can be freely accessed under arternolis.lcsb.uni.lu. The continuous integration framework code is located in the .ci directory and at the root of the repository freely available under github.com/opencobra/cobratoolbox.

Contact: ronan.mt.fleming@gmail.com

Supplementary information: Supplementary data are available at Bioinformatics online.

1 Introduction

Implementation of measures to ensure reproducibility of computational analyses is a fundamental aspect of scientific credibility (Baker et al., 2016; Munafò et al., 2017), in particular in computational biology (Beaulieu-Jones et al., 2017). Analysis software developed collaboratively offers the potential for synergistic effort, but is prone to instability over time due to varying degrees of specialist domain knowledge of code contributors, laborious manual integration of individual code contributions, misinterpretation of the intended operation of code or, especially in academia, the lack of personnel continuity. Tracking of versions and merging of code is typically done with version control software, e.g., git (git-scm.com).

The process of integrating individual code contributions can be semi-automated using a continuous integration approach (Duvall et al., 2007), which involves automatically testing proposed software changes before they are considered for merger with the main code base (Stahl et al., 2014).

Continuous integration enables immediate evaluation of the system-wide impact of local code changes and accelerates the development of quality code by enabling the early detection and tracking of errors, defects, or compatibility issues that can arise when a proposed software change undermines some previously established functionality in an unanticipated manner.

Continuous integration systems, such as Travis (travis-ci.org) or Jenkins (jenkins.io), are tailored to continuously integrate code and are commonly used for testing open-source code written in Python (Ebrahim et al., 2013) or Julia (Heirendt et al., 2017) on a publicly available infrastructure. The COBRA Toolbox is a collaboratively developed software tool for creation, analysis and mechanistic modelling of genome-scale biochemical networks (Heirendt and Arreckx et al., 2017). It is distributed as open-source code (github.com/opencobra/cobratoolbox), but as it is dependent on MATLAB (The Mathworks, Inc.), using public infrastructure for continuous integration is hardly possible. There is a need for a customisable and fully-controllable environment that offers several types of continuous integration jobs, allows for 20 or more concurrent builds, is expandable, permits cross-repository testing on multiple operating systems (including macOS), and that satisfies short and long-term financial constraints.

This need is apparent when aiming at integrating publicly available code that is dependent on licensed software, such as MATLAB, and even more so in regards to technical challenges, such as the activation of licensed software or the size of proprietary and licensed software container images.

Here we present a general purpose infrastructure software application for continuous integration that is compatible with dependencies on licensed software. We illustrate its utility for development of software within the COBRA Toolbox, but ARTENOLIS is ready to be used with other tools using licensed dependencies. This approach ensures that existing high-quality COBRA Toolbox code is stably maintained to ensure reproducibility, yet permits the rapid integration of improvements to existing methods as well as the addition of novel COBRA methods by an active and geographically dispersed openCOBRA development community (opencobra.github.io).

2 Implementation

ARTENOLIS shown in Figure 1 ensures that the rarely smooth and seamless process of integrating individual contributions, the so-called Integration Hell, is avoided.

The implementation of ARTENOLIS includes the configuration of Jenkins and its slaves, the continuous integration code that couples ARTENOLIS to the repository, and customisation code for repository testing and code quality evaluation. Key to the present multi-lingual implementation are the unique cross-platform triggering mechanism, the
The core of the infrastructure is **Jenkins**, the open source automation server for continuous delivery (Ferguson Smart *et al*., 2011). The continuous integration infrastructure consists of a cascade of 3 distinct layers and is a master-slave architecture: a public interaction layer (top), the Jenkins server layer, and the layer with 4 computing nodes behind a firewall, each running a different operating system. A change made in a public repository on **GitHub** (github.com) is seen by the Jenkins server (master), which in turn triggers multiple builds on the 4 computing nodes simultaneously (slaves) (see the Supplementary Information section for details).

Multiple versions of MATLAB and dependencies, and the customised and on-the-fly tutorial and documentation deployment.

Prompt feedback on the quality and the stability of the submission is provided through a comprehensive console output and build badges before the merger of a code contribution and the evaluation of the actual stability and the deployment of the documentation. The tutorials are automatically and regularly tested to ensure error-free execution with the fast developing code base.

### 3 Conclusions

**ARTENOLIS** offers a flexible continuous integration solution for development of any open-source software with licensed dependencies. Applied to software development in the **COBRA Toolbox**, it ensures the stability required for reproducibility of research results yet accelerates the evolution of software implementations of novel constraint-based modelling methods.

### Acknowledgements and funding

The authors acknowledge the support from Thomas Pfau for extensively testing the continuous integration server. The Reproducible Research Results (R3) team of the Luxembourg Centre for Systems Biomedicine is acknowledged for support of the project and for promoting reproducible research.

**ARTENOLIS** was funded by the National Centre of Excellence in Research (NCER) on Parkinson’s disease, the U.S. Department of Energy, Offices of Advanced Scientific Computing Research and the Biological and Environmental Research as part of the Scientific Discovery Through Advanced Computing program, grant no. DE-SC0010429, and the Luxembourg National Research Fund (FNR) ATTRACT program (FNR/A12/01) and OPEN (FNR/O16/11402054) grants.

### Conflict of Interest

None declared.

### References

Baker, M. *et al*. (2016) 1,500 scientists lift the lid on reproducibility, *Nature News*, 533, 452–454, doi:10.1038/533452a.

Beaulieu-Jones, B. K. *et al*. (2017) Reproducibility of computational workflows is automated using continuous analysis, *Nature Biotechnology*, 35, 342–346, doi:10.1038/nbt.3780.

Duvall, P. M. *et al*. (2007) *Continuous Integration: Improving Software Quality and Reducing Risk*, Pearson Education, Edition 1.

Ebrahim, A. *et al*. (2013) COBRApy: COnstraints-Based Reconstruction and Analysis for Python, *BMC Systems Biology*, 7, 74–79, doi:10.1186/1752-0509-7-74.

Ferguson Smart, J. *et al*. (2011) Jenkins: The Definitive Guide: Continuous Integration for the Masses, O’Reilly, Edition 1.

Heirendt, L. *et al*. (2017) DistributedFBA.jl: high-level, high-performance flux balance analysis in Julia, *Bioinformatics*, 33, 1421–1423, doi:10.1093/bioinformatics/btw838.

Heirendt, L. and Arreckx, S. *et al*. (2017) Creation and analysis of biochemical constraint-based models: the COBRA Toolbox v3.0., arXiv:1710.04038 [q-bio.QM].

Munafò, M. R. *et al*. (2017) A manifesto for reproducible science, *Nature Human Behaviour*, 1, 1–9, doi:10.1038/s41562-016-0021.

Stähle, D. *et al*. (2014) Modeling continuous integration practice differences in industry software development, *Journal of Systems and Software*, 87, 48–59, doi:10.1016/j.jss.2013.08.032.
Detailed continuous integration setup
Supplementary information

S1 Background and overview
Software engineers in industry routinely apply continuous integration approaches to ensure that code developed in a collaborative way results in stable and cross-platform compatible software products (Silver, 2017) and to shorten turnaround times. Continuous integration is used more and more often in computational biology, especially for reproducing data-driven research results (Garjio et al., 2013; Narayanasamy et al., 2016), but rarely when software has a dependency on commercial or licensed software.

Continuous integration practices are adopted at a different pace for industrial and open-source software projects as both face different challenges in software development (Holck et al., 2003). Typically, in a research environment, code might not be used beyond the scientist who authored the code. Researchers focus on obtaining results swiftly rather than writing prototype code that is compatible with several licensed software versions or operating systems. A common practice is to store multiple versions of code on various media with regular backups. This setup eventually works well for a small code base used by a single user, but is prohibitive in a collaborative environment, and more so when intending to guarantee reproducibility of results and aiming to provide a stable code base.

According to Duvall et al. (2007) and Fowler (2006) who analysed similar situations, a continuous integration system should centre itself in a development process aiming for stability, high quality, reproducible results, bug-free code execution, and a pleasant end-user experience. Metrics for code quality, such as code grade or code coverage, help developing functional, high-quality code, and ultimately, code that ensures reproducibility of results (see Section S7).

For growing projects, scaling up the continuous integration system could be envisaged (Meyer, 2014). Nonetheless, while the code repositories that are continuously integrated must be maintained, the same holds for a running environment for reproducibility and testing (Rogers, 2004).

Integrated software development and analysis environments, such as MATLAB, make it straightforward to start coding and modify existing code, even for novice users. Within the COBRA community, end users run their code on a variety of operating systems, such as Windows, macOS, or Linux. ARTENOLIS benefits all end users considerably by ironing out incompatibilities and defects, especially Windows users. The underlying server architecture is explained in detail in Section S3 and the configuration of ARTENOLIS is described in Section S4.

S2 Cascade of the Continuous Integration system
The cascade shown in Figure 1 is typical of a continuous integration system of a reproducibility and testing infrastructure, but includes specificities tailored to the COBRA Toolbox.

Public interaction layer
The main code base, as well as all of its forks, are located on GitHub, e.g., the repository of the COBRA Toolbox is located at github.com/opencobra/cobratoolbox. A fork is an individual and modifiable copy of the main code base of the contributor with <userName>, and is located under github.com/<userName>/cobratoolbox. Contributors may contribute to the main code base by opening a pull request (PR), which is reviewed and tested. In particular, developers of the COBRA Toolbox repository may easily submit pull requests using the MATLAB.devTools (github.com/opencobra/MATLAB.devTools).

The public interaction layer also includes the badges of the build status (see Section S5.2) and the publicly accessible website opencobra.github.io/cobratoolbox, which hosts the documentation and tutorials. The details on the generation and deployment of the documentation are provided in Section S6. The repository of the COBRA Toolbox is structured according to common practices of continuous integration (e.g., github.com/JuliaLang/julia) and is explained in detail in Section S4.1.

Jenkins server layer
The main element of the Jenkins server layer is the Jenkins software installed on a virtual machine, which acts as a master node. The Jenkins server is accessible from the Internet and is constantly listening for activity on the main GitHub repository via web-hooks. Once a PR is submitted by a contributor, builds are triggered on the slave machines through secure Java Web Start communication.

The Jenkins server layer partially includes the codecov and Documenter.py elements. The coverage report is prepared on the Jenkins server, and is being made publicly available through codecov under codecov.io/github/opencobra/cobratoolbox. Similarly, the documentation is generated on the Jenkins server before being deployed to the opencobra.github.io/cobratoolbox website.

Computing nodes layer
The computing nodes layer contains 4 computing nodes, each running a different operating system. Most importantly, the continuous integration system set up for the COBRA Toolbox runs on the most popular operating systems in the COBRA community: Linux, macOS, Windows 7, and Windows 10. A virtualisation environment bears key advantages for a continuous integration system and is described in Section S3.

When triggered, each of the 4 computing nodes launches the last 4 stable versions of MATLAB (R2014b, R2015b, R2016b, and R2017b). Each MATLAB version then runs the dedicated testing suite (see Section S5.1), which makes use of multiple solvers, such as CPLEX (IBM ILOG, 2017), Gurobi Optimization (2017), Tomlab (Holmström et al., 2004), Mosek (Andersen et al., 2000), GLPK (GNU Linear Programming Kit), PDCO (Chen et al., 1998) and DQQ (Ma et al., 2017).

S3 Architecture
S3.1 Computing nodes and resources
The advantage of virtual machines over physical machines is that various computing environments can co-exist on a single physical machine. A
setup with fewer physical machines but with multiple virtual environments is more economical and occupies a smaller space in the server racks. Thanks to a virtualisation layer and a hypervisor monitoring the health (see Section S3.3), new virtual machines can be created or deleted on demand and based on the available capacity of the physical server. The Jenkins virtual machine running the Linux Ubuntu operating system is a shared resource, is referred to as the master node, handles HTTPs requests, and orchestrates the slave nodes.

The Linux (Debian based), Windows 7, and Windows 10 operating systems are virtual machines and are running on the same physical computing node, whereas the macOS operating system is running on a dedicated physical machine. The specifications of the computing nodes are provided in Table S1. A limited amount of storage is usually provided to each virtual machine and only satisfies the need to run and store small amount of data. For larger data, such as build data, an NFS (Network File System) or SMB (Server Message Block) mount point is provided on a central storage system.

### S3.2 Virtual machine management and access control

Security and appropriate access control to each virtual machine and associated services is provided through FreeIPA (freeri.org). FreeIPA provides a centralised resource to control authentication, authorisation and account information by storing all information about a user, virtual machine and other sets of objects required to manage the security of the virtual machine.

In order to reduce maintenance and initialisation time of new virtual machines, the virtual machines are deployed using Foreman (theforeman.org), a virtualisation environment agnostic web tool typically used to manage the complete lifecycle of virtual machines.

All physical and virtual servers (except the macOS node) are configured via Puppet, which is a configuration management tool that facilitates standardised configurations across a pool of servers such as firewall definitions, administration SSH (Secure Shell) keys, default packages, and other settings. In the current setup, the compliance of all machine configurations is constantly monitored, while periodic health reports are provided to Foreman. Besides the configuration monitoring and deployment, the performance of the virtual machines is monitored using netdata (github.com/firehol/netdata), which is particularly useful when evaluating the performance of the continuous integration test suite (see Section S4.4).

### S3.3 Virtualisation layer

Hypervisor software runs on a server handling virtual machines (VMs). In the present case, oVirt (ovirt.org) is installed on each of the physical servers, and is a key element in VM management. Kernel-based Virtual Machine (KVM), a virtualisation infrastructure, is used as the hypervisor layer. oVirt provides a graphical user interface (GUI) to manage all physical and logical resources needed for the virtual infrastructure (e.g. storage, network, data centres). In the current continuous integration system, the 2 physical servers (not the macOS node, see Table S1) are running CentOS 7.3 and are virtualised using oVirt 4.1.0.

### S4 Configuration ARTENOLIS

#### S4.1 Repository structure and branches

In Table S2, the main directories of the COBRA Toolbox repository are listed together with their respective purpose. The test suite is located in /test, while the source files of the code base are located in the /src directory. These two directories, together with /tutorials, are key components. The /ci directory contains scripts required for the continuous integration integration.

The scripts travis.yml and codecov.yml located at the root of the repository are required to trigger the jobs (see Section S4.5) and to report code coverage (see Section S7.1), respectively.

#### Directory/file name | Purpose
--- | ---
/ci | Directory with continuous integration bash-scripts
/src | Directory with code source files
/test | Directory with test files and testAll.m
/tutorials | Directory with tutorial .mlx files
/travis.yml | YAML trigger script
/codecov.yml | YAML script for code coverage report

Table S2. Structure and key directories and files of the COBRA Toolbox repository.

As explained in Section S4.4, the COBRA Toolbox follows the common development model of a stable master branch and a develop branch for development. This development model is particularly well suited for a reproducibility and testing infrastructure, such as ARTENOLIS. It is against the develop branch that new pull requests are raised by developers, while it is the master branch that contains the stable version of the COBRA Toolbox. A regular merge strategy from the develop to the master branch ensures that the latest features are adopted in the stable version. As the development of new features is made on separate branches that are being merged to the develop branch only through Pull Requests (PRs), and as each PR is being tested by the continuous integration system, the risk of the develop branch failing to build is very low. This stability of the master branch is particularly important in a fast-moving research environment that relies on the reproducibility of data-driven results.

#### S4.2 Local and GitHub user accounts

A local non-administrative user account on the slave nodes is set up in order to run MATLAB as a specific user during a continuous integration build.
have proper read/write permissions, and allow for simplified repository maintenance and/or debugging. This user is not bound to a physical administrator, and can be associated to licenses or perform repetitive tasks, such as setting the build badges (see Section S5.2). This local user account is also not intended to be used to perform administrative tasks on the computing node. Within ARTEMOLIS, this local user account is named jenkins, and is independent of server-wide access control as explained in Section S3.2.

In order to perform certain repository related tasks, such as deploying the documentation (see Section S6), a dedicated GitHub account that is not related to a physical person has been created. This GitHub account is a bot-type account, named cobrabort, and is only used for administrative tasks or committing/pushing automatically to the GitHub server.

### S4.3 master and slave nodes

The configuration of the continuous integration system exploits the master/slave functionality integrated in Jenkins. A large number of jobs running on various operating systems (see Section S4.4) may be triggered from a single Jenkins server. The workload of multiple jobs can be delegated by the master node to multiple slave nodes, which allows for a single Jenkins installation. The master node serves the web interface of Jenkins (artenolis.lcsb.uni.lu) and acts as a portal to the entire farm of slave nodes. The load of the jobs is distributed on the master node, while the slave nodes are triggered accordingly.

The slaves are connected to the master node through Java Web Start (Java Network Launch Protocol - JNLP). An agent is running on the slave that listens for a triggering signal from Jenkins on the master node. Once triggered, the slave node executes the job and reports back to the master node a build status as explained in Section S5.2. Importantly, on all computing nodes, the service running the agent must be configured to launch upon startup of the server, as Jenkins on the master node is not able to wake the agent on the slaves. On macOS, it is important to run Caffeine (lightheadsw.com/caffeine), a tiny program that prevents the macOS system from automatically activating the sleep function.

### S4.4 Job definitions

A job is a configuration of a build pipeline on the Jenkins server, has a specific purpose, runs on a different slave/operating system, or builds a different branch of the repository. An example of such a configuration file is given under artenolis.lcsb.uni.lu/userContent/configExample.yml. The job definitions in Jenkins on the master node are configured accordingly as indicated in Table S3.

| Job name | Description | R2017b | R2016b | R2015b | R2014b |
|----------|-------------|--------|--------|--------|--------|
| COBRAtoolbox-branches-auto-linux | Build the develop and master branches (automatic trigger) | ⋆ | ⋆ | ⋆ | ⋆ |
| COBRAtoolbox-branches-auto-macos | Build the develop and master branches by SHA1 (Secure Hash Algorithm 1) (manual trigger) | * | * | * | * |
| COBRAtoolbox-branches-auto-windows7 | Build a Pull Request by SHA1 (manual trigger) | * | * | * | * |
| COBRAtoolbox-branches-auto-windows10 | Request | * | * | * | * |
| COBRAtoolbox-branches-manual-linux | Build any newly submitted Pull Request | ⋆ | ⋆ | ⋆ | ⋆ |
| COBRAtoolbox-pr-auto-linux | | * | * | * | * |
| COBRAtoolbox-pr-auto-macos | | * | * | * | * |
| COBRAtoolbox-pr-auto-windows7 | | * | * | * | * |
| COBRAtoolbox-pr-auto-windows10 | | * | * | * | * |

Table S3. Job definitions for the continuous integration setup of the COBRA Toolbox repository.

In essence, one job is defined per operating system, and in this particular continuous integration setup, one job per slave. This setup has been chosen in order to provide the highest robustness of ARTEMOLIS. A Jenkins job may also be parameterised. In the case of the COBRA Toolbox repository, a matrix of sub-jobs is generated for different MATLAB versions using the MATLAB_VER parameter.

In order to streamline the continuous integration setup, there are two distinct job types: jobs that trigger builds of the develop and master branches (marked with --branches) and jobs marked with --pr that build any newly submitted pull request (PR).

Each job type is either triggered automatically by Jenkins or manually by an administrator using the SHA1 of the commit. As the develop and master branches must be tested for all supported MATLAB versions, the --branches jobs run for each supported MATLAB version on each slave. However, whenever a Pull Request is submitted, only the job running on the Linux node triggers the test suite on all supported MATLAB versions. On all other slaves (macOS, Windows 7, and Windows 10), only the most stable and most used version of MATLAB is tested. This setup ensures a long time after the release of a new MATLAB version. These release schedule, and the compatibility of certain solvers is generally only ensured a long time after the release of a new MATLAB version. These combinations are prone for incompatibilities. The compatibility of the COBRA Toolbox has been tested and evaluated separately such that incompatibility issues between solver versions, operating systems, and MATLAB versions do not crash on the continuous integration server or on the user system. A solver compatibility matrix has already been established for all supported operating systems and actively supported solver interfaces with...
their respective versions. This compatibility matrix is used by MATLAB during runtime to determine whether a certain user setup is compatible or not ([opencobra.github.io/cobratoolbox/docs/compatibility.html]), and ensures that certain tests on the continuous integration setup are skipped, which would otherwise lead to a build failure.

Each job can either succeed or fail. The build stability of each job and build trend is monitored and can be retrieved from [artenolis.lcsb.uni.lu/job/<jobName>/buildTime-Trend](http://artenolis.lcsb.uni.lu/job/<jobName>/buildTime-Trend), where `<jobName>` is the name of a job as listed in Table S3. Currently, a job on the continuous integration system takes in average around 30-40 minutes to finish (all MATLAB versions). Although all MATLAB versions are launched in parallel, and despite some tests of the test suite requesting a parallel pool of workers, the memory consumption is moderate. Each job requests about 12GB of memory, while all CPUs of the virtual machines are used of up to 60%. For each job, input/output (IO) on the slaves is high at the beginning during the repository cloning phase, but is negligible during the test run itself. If more CPU power or RAM was needed in the future, the technical characteristics of the virtual machines may effortlessly be changed, which is another advantage of the master-slave setup and of the virtual machines as explained in Sections S4.3 and S3.1. As shown in Table S1, the hard-drive of each of the slaves has limited capacity. All builds are stored on a central storage server, and old builds are discarded. The workspace itself is cleaned after each job in order to reduce the storage needs.

The performance of the slaves and of the hypervisor are monitored internally using netdata ([github.com/firehol/netdata]). Over time, the Jenkins memory usage on the master node increases gradually. In order to avoid the master node to swap memory, Jenkins is restarted every night when being idle. A regular restart also ensures that the configuration files of Jenkins are reloaded and that the latest configuration files of Jenkins and the jobs are used.

**S4.5 Continuous integration scripts**

When triggered, each job starts by cloning the repository, and checks out the latest commit or the commit that shall be built. On the Jenkins node, the Java web call triggers the interpretation of a YAML (YAML Ain’t Markup Language) script, namely the `travis.yml` script shown in Listing S1. Together with build-specific environment variables, an executable shell script, also known as the Hudson shell file, is generated. This Hudson shell file is then sent to and executed on each slave in a shell-like environment. The continuous integration process is consequently started through the `travis.yml` script placed at the root of the repository directory.

**Listing S1.** `travis.yml` script interpreted by Jenkins and used to trigger the `runtests.sh` script.

```bash
language: bash

before_install:
  # fresh clone of the repository
  - if [[ -a .git/shallow ]]; then
    git fetch --unshallow;
  fi

script:
  # launch the tests
  - bash .ci/runtests.sh
```

As a shell script offers cross-platform flexibility, a proper shell script is called from the YAML script, namely `runtests.sh` located in the specific `.ci` folder. The shell script `runtests.sh` is shown in Listing S2 and runs on all supported platforms. Each platform is identified by the environment variable `ARCH`, which is set by the job definitions (see Section S4.4). The simplest launch command is set for UNIX operating systems (Linux and macOS).

As explained in Section S4.3, the usage of the tiny caffeine program is obvious when running the script on macOS. On UNIX, any output of MATLAB while running the `testAll.m` script is routed directly to the shell of the slave, and ultimately, to the console publicly accessible on Jenkins under [artenolis.lcsb.uni.lu/job/<jobName>/buildNumber/MATLAB_VER=<version>/label=<platform>/console](http://artenolis.lcsb.uni.lu/job/<jobName>/buildNumber/MATLAB_VER=<version>/label=<platform>/console), where `<jobName>` is the name of the job as defined in Table S3, `<buildNumber>` is the number of the build, `<platform>` is the label of the operating system, and `<version>` is the MATLAB version.

A key challenge with setting up ARTEONLIS is to trigger a build on a Windows platform and launch MATLAB while providing live feedback, similar to UNIX platforms. As no native `bash` console exists on DOS-based systems, the output of MATLAB is not directly routed to the console. Instead, `git Bash (git-scm.com/download/win)` is used, and the Hudson script launching MATLAB is executed in `sh.exe`. As the output from a shell-like environment is not routed back to the master node directly, a computational trick must be used in order to however display a live console output on the Jenkins web interface. This trick ensures the homogeneity of ARTEONLIS despite the large differences between DOS and UNIX operating systems supported by ARTEONLIS.

The output of MATLAB is routed to a file (`output.log`), while the process is run as a hidden process marked with `i`. The process ID is saved as `PID`. While the hidden process is running, the log file is constantly read in (or followed) by the system command `tail`, whose output is redirected to Jenkins. The shell script `runtests.sh` then only exits once the MATLAB process with `PID` has been executed without an error.

Any eventual error code thrown during this process is caught in the variable `$CODE`. This is the exit code of the script `runtests.sh` that is returned to Jenkins running on the master node. More details on how this feedback code is interpreted are given in Section S5.2.

**S4.6 GitHub interaction**

In order for a build to be triggered from GitHub on the Jenkins server, a so-called `web-hook` has been installed on GitHub. This hook listens to events that occur on the GitHub repository of the COBRA Toolbox and include the opening of a pull request, a change of a commit, or another status modification. On Jenkins, a cookie is stored, which allows Jenkins to listen to that particular hook and that cannot be triggered by any other hook.

A valuable feature of the continuous integration system is that the continuous integration software Jenkins integrates seamlessly with the version control server GitHub. Next to each commit that has triggered a build on the continuous integration system, a status is displayed (see Section S5.2). This visual system is standard for continuously integrated repositories on GitHub, and allows developers to swiftly check and track the status of a build from GitHub directly.

**S5 Evaluation of code stability**

**S5.1 Continuous integration test suite**

The test-suite relies on testing framework functions and is designed to test the functionality and performance of the MATLAB code of the COBRA Toolbox. Once MATLAB is running on the continuous integration server, the test-suite is launched from the `testAll` script. The test-suite makes use of the dedicated unit testing functions implemented
in MATLAB. The testAll script is structured such that MOcov and JsonLab are added to the path first, global variables are defined, and the code quality grade is determined (see Section S7.2) before running any unit-test functions. The runtests command runs in serial (or in parallel) all test files in the test folder. Once the runtests command completed, the number of tests that failed and/or are incomplete is evaluated, and the code coverage percentage is computed (see Section S7.1 for details).

A test, part of the continuous integration test suite, is tailored to run a function in the src directory such that a result is output, a warning, or an error are thrown. A test must evaluate the result returned by the function against a pre-computed reference result. This evaluation within the test is performed using the assert function. All tests for the COBRA Toolbox follow a template, and a guide helps developers get started (opencobra.github.io/cobratoolbox/docs/contributing.html).

The runtests command prints a table with the test name, the running time of each test, and whether the test failed or succeeded. As the exit status code returned by runtests determines the exit code of the MATLAB process (see Section S4.5), the continuous integration test suite is launched within a try...catch statement. The exit code is explicitly set to 1 if the number of failed or incomplete tests is not 0. The exit(exit_code) command is called on the continuous integration server before the end of the try...catch statement. Any exception thrown, such as a crash of MATLAB, is caught in the catch statement. On the continuous integration server, this leads to a returned exit code of 1, while on a user computer, the exception is explicitly rethrown.

The core of the continuous integration system is the test suite. In other words, the number and quality of tests determine the quality of the code in the src directory. Writing tests may be a long process, in particular for bug prone code. In the case of the COBRA Toolbox, the community picked up writing tests for their own functions, leading to a steady increase of code quality (see Section S7).

S5.2 Continuous integration feedback

An execution of a job on the continuous integration server, or a run of the MATLAB test suite, is considered as a build. For each commit on the develop and master branches, a job- and build-specific build status is set on GitHub and updated continuously while the build is running. This commit build status reflects the actual status of the build on the server. The build status is either a green check mark for success, a red cross mark for failure, or a yellow dot for a job that is pending. A commit can hence cause the failure or the success of a job. The trigger build status, or global build status, is set as successful when all jobs ran without errors on a certain operating system.

Commonly, one or multiple visible badges are displayed visibly on the first page of a GitHub repository that indicate the status of the latest builds of the COBRA Toolbox for each MATLAB version, or a build status for a job that is pending. A commit can hence cause the failure or the success of a job. The trigger build status, or global build status, is set as successful when all jobs ran without errors on a certain operating system.

For the case of the COBRA Toolbox repository, the MATLAB test suite is run on multiple operating systems. As for each build (i.e., for each MATLAB version), a build status is returned, a matrix of build status is defined, which can be consulted under opencobra.github.io/cobratoolbox/docs/badges.html. In order to simplify the readout for the end-user who is primarily interested in the stability and reproducibility of the development of a repository.

For each commit on the COBRA Toolbox repository, the MATLAB test suite is run on multiple operating systems. As for each build (i.e., for each MATLAB version), a build status is returned, a matrix of build status is defined, which can be consulted under opencobra.github.io/cobratoolbox/docs/badges.html. In order to simplify the readout for the end-user who is primarily interested in the stability and reproducibility of the development of a repository.
Another valuable feedback mechanism is to alert the administrator of the continuous integration system of build failures by email. Other health monitoring tools, such as explained in Section S3.2, help determine the cause of failure in cases where the failure of a build might be system related (e.g., high memory usage or faulty disks).

S6 Documentation and tutorials

The documentation of the COBRA Toolbox is generated and deployed automatically once a push is made on the master or develop branch of the GitHub repository, and if the build is successful.

The building of the documentation is done in 3 phases: the generation of the tutorials, the generation of the documentation and its deployment to a host server. Figure S2 represents all the steps involved in the creation and deployment of tutorials and documentation, and summarises the creative pipelines.

As MATLAB does not provide a full documentation system to publish documentation of user created MATLAB codes, a specific documentation pipeline that uses the popular Python documentation tool Sphinx (sphinx-doc.org) has been developed. Documenter.py (github.com/syarra/Documenter.py) is at the heart of the documentation pipeline. The package is designed to combine reStructuredText files and inline docstrings from MATLAB functions into a single interlinked publishable documentation. A docstring is a function is a string literal specified in source code that is used to document a function in an easily comprehensible way. In order for the documentation to be generated properly, the docstrings must be written using keywords listed in Table S4. A keyword defines the start of a block of documentation with the header of a function, must be followed by non-empty lines, and is separated from the next block by an empty line.

| Keyword | Purpose |
|---------|---------|
| USAGE: | defines how to use the function |
| INPUT: or INPUTS: | describes input argument(s) |
| OUTPUT: or OUTPUTS: | describes the output argument(s) |
| EXAMPLE: | shows example of code (MATLAB syntax) |
| NOTE: | displays a highlighted box with text |
| AUTHOR: | lists author(s) of the function |

Table S4. Main keywords in the function headers (docstrings) used for documentation extraction.

The first phase in the generation of the full documentation is the generation of the documentation of user created MATLAB functions (steps I.a-I.d in Figure S2). MATLAB provides a very convenient way of writing tutorials using Live Script, which is a document containing both computer code and text elements, such as paragraphs, equations, figures, links, and which is saved as a .mlx file. In order to allow for swift consultancy of the tutorial through a web browser, or even print a hard copy of the tutorial, the Live Scripts are converted automatically into PDF and HTML formats by executing the prepareTutorials.sh script (steps I.a and I.b) based on MATLAB functionality. During steps I.e and I.d, the .pdf and .html versions of the tutorials are further modified to fit the webpage style and moved to the web server location. A user running MATLAB R2016a or above may consult the tutorials in 3 different formats: as a document (.pdf), on the web (.html), or locally directly in MATLAB (.mlx). The Live Script functionality is not available for versions of MATLAB older than R2016a. For convenience, the .pdf documents are converted to .png image files and displayed directly within the README.md files on GitHub when browsing the /docs/source/modules/*.rst versions of the files.

The second phase consists of extracting the docstrings from MATLAB functions using the Sphinx package and the matplotlib plugin (pypi.python.org/pypi/sphinxcontrib-matplotlib) (steps II.a-II.d) based on the docstrings and the keywords shown in Table S4. The docstrings are then combined with the reStructuredText files located in the /docs/source/modules directory from the original repository to produce HTML pages (steps II.e-II.h) that can be displayed on the web server. Style and layout are matched to the style of the web documentation for Julia packages thanks to the Sphinx COBRA Theme package (github.com/uni-lu/sphinx_cobra_theme). The Julia Sphinx theme (github.com/uni-lu/sphinx_julia_theme) is adopted in order to provide a harmonised package suite together with COBRA.jl (Heirendt et al., 2017) (opencobra.github.io/COBRA.jl/stable).

The third and last phase of the documentation generation is the deployment to the host web server (opencobra.github.io/cobratoolbox) via the gh-pages branch of the COBRA Toolbox repository (github.com/opencobra/cobratoolbox/tree/gh-pages). As explained in Section S4.2, the user cobrabot pushes the newest changes and publishes the latest version of the documentation.
S7 Evaluation of code quality

S7.1 Code coverage

Functional coverage provides information about which scenarios have been tested. In software development, it is essential to track test coverage, or in other words, which functions and source lines are executed during the test run. The coverage report helps to estimate how much of the code base is tested or executed without crashing and how much code is not covered through testing. Code coverage is reported through a coverage report generator for MATLAB and GNU Octave, namely the MOCoV toolbox (github.com/MOCov/MOCov), and through the free and GitHub integrated codecov.io service. A code coverage report reveals which lines have been added and whether they have been tested. The code coverage difference can consequently be determined for every pull request.

The code coverage determines the scope of the test suite and is determined as a ratio of the number of executed code source lines and the total number of executable lines of code. Executable lines of code are counted as all lines in the .m files in the */src* folder that do not start with % or the language keywords *end*, *otherwise*, *switch*, *else*, *case*, or *function*.

Tracking the code coverage provides a measure of stability of the code base and the breadth of the test suite. The limitations of code coverage reports include that the intended functionality of the code is not verified and that the quality of the code is not assessed. Although there is no precise measure for code quality itself, the efficiency of code certainly can be graded (see Section S7.2).

S7.2 Code efficiency grade

The code efficiency grade is a valuable measure of how MATLAB code can be improved for efficiency and to detect potential problems and opportunities for code improvement. The implemented function checkcode is run on each source code file and the number of MATLAB Code Analyzer messages is recorded.

| Code efficiency grade | Percentage range |
|-----------------------|------------------|
| A                     | 0% - 3%          |
| B                     | 3% - 6%          |
| C                     | 6% - 9%          |
| D                     | 9% - 12%         |
| E                     | 12% - 15%        |
| F                     | > 15%            |

Table S5. Conversion chart from coverage percentage to quality grade.

The average number of messages per source code file is divided by the actual number of executable source code lines, which yields the code grade percentage. For ease of use, the code grade percentage is converted to a letter code as shown in Table S5.

S7.3 Code linting

Linting is the practice of harmonising a given code style across an entire code base. The need for code linting of an open-source code base is obvious; many developers contributing to the same project and source files may not have the same discipline of writing code according to style guidelines.

A clean, homogeneous, and easily readable code base is required for easy debugging and swift maintenance. Automatlab.py (github.com/syarra/automatlab) is a tool designed to ensure that MATLAB code comply to style conventions defined at opencobra.github.io/cobratoolbox/docs/styleGuide.html.

Automatlab.py runs in two steps: first, a compliance analysis is performed to check the adherence to the style guide. Then, formatting is fixed directly in the MATLAB code.

Linting is a common practice when coding using Python (e.g., the package AutoPEP8: github.com/hlato/autopep8) or Julia (e.g., the package Lint.jl: github.com/tonyhffong/Lint.jl). Several state-of-the-art editors, such as Atom (atom.io), use built-in linting functionality to advise the developer of best coding practices.

List of Acronyms

| Acronym | Designation |
|---------|-------------|
| ARTENOLIS | Automated Reproducibility and Testing Environment for Licensed Software |
| COBRA | COntstraint-Based Reconstruction and Analysis |
| HTTP | Hypertext Transfer Protocol |
| JNLP | Java Network Launch Protocol |
| KVM | Kernel-based Virtual Machine |
| NFS | Network File System |
| PR | Pull request |
| SHA1 | Secure Hash Algorithm 1 |
| SMB | Server Message Block |
| SSH | Secure Shell |
| YAML | YAML Ain’t Markup Language |

References

Andersen, E. D. et al. (2000) The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm, *High Performance Optimization*, 33, 197–232, doi:10.1007/978-1-4757-3216-0.

Chen, S. S. et al. (1998) Atomic Decomposition by Basis Pursuit, *SIAM Journal on Scientific Computing*, 20, 33–61, doi:10.1137/S1064827596304010.

Duvall, P. M. et al. (2007) Continuous Integration: Improving Software Quality and Reducing Risk, Pearson Education, Edition 1.

Fowler, M. (2006). Continuous Integration, Article on martinfowler.com.

Garro, D. et al. (2013) Quantifying Reproducibility in Computational Biology: The Case of the Tuberculosis Drugome, *Plos One*, 8, doi:10.1371/journal.pone.0080278.

Gurobi (2017), Documentation (Gurobi Solver), Technical Documentation.

Gurobi (2017), Documentation (Gurobi Solver), Technical Documentation.

Heirrendt, L. et al. (2017) DistributedFBA.jl: high-level, high-performance flux balance analysis in Julia, *Bioinformatics*, 33, 1421–1423, doi:10.1093/bioinformatics/btw838.

Holck, J. et al. (2003) Continuous Integration and Quality Assurance: a case study of two open source projects, *Australasian Journal of Information Systems*, 11, doi:10.3127/ajis.v11i1.145.

Holmstrom, K. et al. (2004) *The TOMLAB Optimization Environment, Modeling Languages in Mathematical Optimization*, Springer, 369–376, doi:10.1007/978-1-4613-0215-5.

IBM ILOG (2017), Documentation (CPLEX Optimization Studio V12.7.1), Technical Documentation.

Ma, D. et al. (2017) Reliable and efficient solution of genome-scale models of Metabolism and macromolecular Expression, *Scientific Reports*, 7, doi:10.1038/srep08683.

Meyer, M. (2014) Continuous Integration and Its Tools, *IEEE Software*, 31, 14–16, doi:10.1109/MS.2014.58.

Narayanasamy, S. et al. (2016) IMP: a pipeline for reproducible reference-independent integrated metagenomic and metatranscriptomic analyses, *Genome Biology*, 17, 260–281, doi:10.1186/s13059-016-1116-8.

Rogers, R. O. et al. (2004) Scaling Continuous Integration, *Extreme Programming and Agile Processes in Software Engineering*, Springer, 68–76, doi:10.1007/b988510.

Silver, A. (2017) Collaborative software development made easy, *Nature News*, 550, 143–144, doi:10.1038/550143a.