MoCITempGen: Modelica Continuous Integration Template Generator

David Jansen Fabian Wüllhorst Sven Hinrichs Dirk Müller

Institute for Energy Efficient Buildings and Indoor Climate, E.ON Energy Research Center, RWTH Aachen University, Germany, {david.jansen, fabian.wuellhorst, sven.hinrichs, dmueller}@eonerc.rwth-aachen.de

Abstract

Modelica enables an object-oriented approach to model complex systems in product development and research, and, thus, the development of various model libraries. Library development requires collaborative development in a team of multiple developers. A typical challenge in collaborative development, especially in the area of open source, is to create models of uniform quality despite different levels of knowledge among developers. Techniques, such as Continuous Integration (CI) from the field of software development, can help to solve these challenges. However, the adaptation of CI for the area of Modelica model development currently requires the manual creation of complex templates and a high degree of manual configuration. In this paper we present *MoCITempGen*, an open source tool for automated generation of CI structures for Modelica. The tool is succesfully applied on two Modelica libraries to demonstrate its functionality.

Keywords: Continuous Integration, Modelica testing

1 Introduction

With the progressive use of modeling and simulation, both in research and in product development, ensuring the model quality is becoming increasingly important. As the complexity of systems continues to increase, so does the number of model developers contributing to libraries and individual models. The concept of open source can partially address this problem, as the increased reach enables cross-institutional and cross-company collaboration. However, this creates communities of sometimes very different levels of knowledge with regard to modeling, which further complicates compliance with uniform quality criteria. In the university context, where both research assistants and students work together on models, this effect is partially amplified. Continuous Integration (CI) is a technique from software development, more precisely from *DevOps*, that was first described by Grady Booch (Booch 1991) and later defined as one of the 12 principles of extreme programming (Beck 2000). CI processes in the context of modeling aim to assure the quality of models. These processes include testing of the produced code/models in a designated environment. Vasilescu et al. reviewed 246 GitHub projects that use CI, finding that the use of CI increases the quality of repositories in terms of a higher number of reviewed, merged and rejected pull requests (Vasilescu et al. 2015).

As Modelica models and packages are stored as ASCII-files, the usage of *git* platforms like *GitHub* or *GitLab* is strongly recommended when developing Modelica libraries anyway (Gall et al. 2021). Thus, the application of CI to Modelica code is not new. For instance, the well-known modelica-buildings (Wetter et al. 2014) library has an extensive CI structure based on the *BuildingsPy* Python library (Wetter 2019). Over the past few years, we too have built a comprehensive CI structure for the *AixLib* library that partially reuses functions from *BuildingsPy* and combines them with self developed functionality (Maier et al. 2023). However, these approaches are tailored to the corresponding libraries and their application to other Modelica libraries is not straightforward. To overcome this issues, we developed *MoCITempGen*, an open source tool that allows to generate a complete Modelica CI structure based on a few inputs with various testing stages and functions. With the release of the tool, we want to lower the hurdle of applying CI structures to Modelica libraries and thus increase the quality of open source Modelica modeling projects in the long run.

Before describing *MoCITempGen* in Section 3, we review existing CI solutions in Section 2. To demonstrate the usage of *MoCITempGen*, we apply it in Section 4 to the open-source libraries *AixLib* and *BESMod*. Subsequently, we provide a critical discussion about the limitations of the created CI-structure in section 5.

2 State of the Art

Following, we give an overview of common CI hosts and infrastructures (2.1), and show which approaches and solutions for applying CI to Modelica already exist (2.2).

2.1 Common CI hosts

In order to take advantage of CI, an appropriate infrastructure must be used. In the following, we provide a brief overview about three often used systems, how they can be deployed, and what costs they generate.

Travis-CI[1] offers a standalone CI/CD service, that can be

[1] https://www.travis-ci.com/

connected to multiple platforms like GitHub. In the end of 2020, Travis-CI stopped offering free CI-minutes and is now only available via priced plans. Even if Travis-CI is completely open source, there is currently no option to self-host the service, so using it will always generate costs.
GitHub Actions[2] is a service that was launched in 2018. It is directly integrated into GitHub and offers 2000 free minutes of usage per month. There is also the option to add a self-hosted runner to a repository.
GitLab-CI/CD[3] was first released in 2012 and is the inbuilt CI/CD feature of GitLab. GitLab itself can be self-hosted without costs, and the GitLab runner can be self-hosted as well. If self-hosting is not an option for the application, multiple paid plans for GitLab and its runners exist.
With all self-hosted variants, the costs for providing the respective hardware must, of course, be taken into account.

2.2 Available CI for Modelica

Following, we want to give a short overview about uses of CI in context of Modelica and the tools that are developed around it.
Rabuzin et al. introduced a CI workflow for testing Modelica models via OpenModelica and Travis-CI for their power system library OpenIPSL (Rabuzin, Baudette, and Vanfretti 2017). Their workflow includes a checking stage that checks the compliance with the Modelica syntax and a model validation stage that runs the most current model implementation against existing simulation results of the model to verify that results have not changed.
Schoelzel et al. implemented a fine-grained unit and regression test setup to solve the problem of reproducibility in the context of Modelica models for biology using a CI pipeline based on GitHub Actions(Schölzel et al. 2021).
Hugues et al. perform not only testing but also integrate Continuous Deployment (CD) in their pipelines to create digital twins in form of Modelica functional mock ups (FMU) for cyber-physical systems in their project *TwinOps* (Hugues et al. 2022).
The Buildings[4] library uses an extensive approach with a combination of *Travis* CI and *GitHub Actions* (Wetter 2019). GitHub Actions is used to run different scripts which check HTML syntax, model order, experiment setup, existing documentation and much more. As the job's runtime is short, developers get direct feedback on their contributions. *Travis* CI is used to run regression tests for different software (Dymola, OpenModelica, and Optimica). Furthermore, library specific developments such as the control description language or spawn of energy plus are tested. Some of these scripts are used in the IBPSA[5] and IDEAS as well. The regression results have to be generated manually.
The Open Source Modelica Consortium (OSMC) built an open source pipeline to test all Modelica libraries included in their package manager[6]. For this purpose, all models that have an experiment stop time are simulated with OpenModelica and the different process steps are documented and published in an HTML report.
Furthermore, the Modelica Standard Library also uses CI processes to ensure the quality of the models.
Additionally, there are several tools that can be used in a Modelica CI environment. These include, among others, the BuildingsPy python package[7] library, the modelica formating tool modelica-fmt[8], the ModelicaPy python package[9], and the regression test package MoPyRegtest[10]. Although these tools and their applications already demonstrate a certain potential of CI in the context of model development, their use is often limited to the respective libraries. Even if some approaches can be reused, this often means a manual adaptation to the own library. From our point of view, this creates a gap that can be filled with the development of tools that generate CI structures adapted to one's own library on the basis of a few input parameters. Therefore, we present this new tool, which we call *MoCITempGen*, in the following section.

3 Methodology

In the following sections, we first provide a nomenclature about common terms in the context of CI and Modelica for better understanding (section 3.1). After that, we provide a brief explanation about the general structure of CI setup (section 3.2) and then describe the template creation process (section 3.3). Finally, in section 3.4 we describe how the developed CI processes work in detail.

3.1 CI-Nomenclature

Before we describe the template generation and the functionality of the templates, we want to give a short definition of the common terms for CI in the context of Modelica for better understanding. Since *MoCITempGen* is currently based on GitLab-CI, we use the names and terms based on the definition of GitLab[11]. However, most functionalities and terms of CI pipelines are similar across the different services. Table 1 provides an overview of the most important terms for applying CI to Modelica.

Figure 1 shows the CI-setup based on *MoCITempGen*[12] that can be adapted to any Modelica library. The *MoCITempGen* repository holds the template generation

[2] https://github.com/actions
[3] https://docs.gitlab.com/ee/ci/
[4] https://github.com/lbl-srg/modelica-buildings
[5] https://github.com/ibpsa/modelica-ibpsa
[6] https://github.com/OpenModelica/OpenModelicaLibraryTesting
[7] https://github.com/lbl-srg/BuildingsPy
[8] https://github.com/modelica-tools/modelica-fmt
[9] https://github.com/ORNL-Modelica/ModelicaPy/
[10] https://github.com/modelica-tools/MoPyRegtest
[11] https://docs.gitlab.com/ee/ci/pipelines/
[12] https://github.com/RWTH-EBC/MoCITempGen

DOI 10.3384/ecp204521

Table 1. Nomenclature for CI and Modelica terms.

Term	Explanation
`job`	Smallest part that holds definition of what to do
`script`	Section in a `job` definition with actual commands to execute in a `job`
`stage`	Bundles `jobs` for specific use case (e.g. testing) and defines the order to execute them
`pipeline`	Top-level definition of a CI workflow
`runner`	Receives the jobs `jobs` from the CI structure and executes them
`variables`	Central definition of values to use throughout `stages` and `jobs`
`artifacts`	Store results of `jobs` throughout a `pipeline` and after the `pipeline` finished
`rules`	Define if a job should be executed or not
`extends`	Reuse a `job` definition and only overwrite certain parts of it
`include`	Command that allows to reuse the definition of a `job` in different `pipelines`
`library`	Collection of reusable modeling components, such as models, classes and functions
`package`	Hierarchical grouping of related components within a Modelica `library`
`whitelist`	List of models that should be excluded from specific `stages`

tool that needs to be cloned locally and copied to the directory of the users Modelica library once for setup. Subsequently, the template generation will be performed. The generation process itself will be explained in more detail later.

3.2 Setup description

The resulting library related CI-structure of templates can then either be placed directly inside the target Modelica library (dashed lines) or placed outside in a separate repository using the `include` command of GitLab-CI. The way to store the templates mostly influences how easy updates of the templates can be deployed. Storing the templates in a separate repository is recommended for repositories with intensive development, multiple developers and therefore multiple branches. In such repositories, a deployment or update of the CI templates would otherwise require a merge of the updated templates back into every branch, because every branch has its own version of the templates. For smaller repositories with a small amount of developers, storing the templates directly with the Modelica code is acceptable as well.
The GitLab runner executes the stages and jobs defined in the library related CI-structure using the *ModelicaPyCI* package we released. This package holds the functionalities which will be described in the following sections.

If the target Modelica Library repository is hosted in GitLab, the GitLab runner directly interacts with the main repository and the process is completed. However, any other Git provider, like GitHub, BitBucket or AWS CodeCommit can be used by taking advantage of the GitLab mirroring feature. This way, the target Modelica library is mirrored into a separate GitLab repository (dotted objects). For GitHub and BitBucket it is also possible to display the pipeline status in the target Modelica library repository.

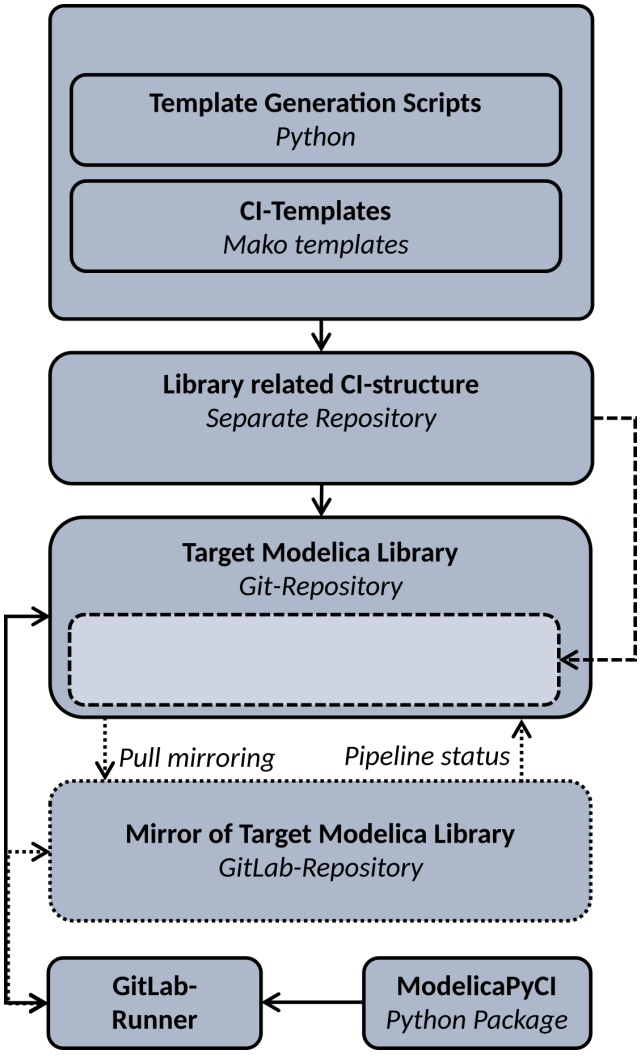

Figure 1. CI setup, scattered objects are optional, dotted objects are only needed if not using GitLab as main repository.

3.3 Templates Generation Process

The template generation is performed via Python using the Python templating package Mako[13]. This allows the dynamic creation of templates based on the respective Modelica library. The generation process needs information about the repository, whether a mirroring process is required, whether certain models should be excluded from the CI process, what steps and tasks should be performed, and whether and how to include manual interaction with the CI.
To get this information, the setup process is possible in two ways. The first option is an interactive way, where the setup process uses command line interaction and checks the file structure and existing Modelica packages based on the library structure first and creates the setup subsequently based on user inputs. The second option is a configuration based way, where the user fills out a `.toml` configuration file in advance. The latter is more suited to apply adjustments to an already generated CI-structure as the interactive process creates the `.toml` configuration file for later adjustments.

3.4 Template Structure and Functionality

Figure 2 shows a simplified structure of the templates generated by *MoCITempGen*. The created template structure consists of single template file for each stage, which are combined in a top level `gitlab-ci.yml` file. Some jobs are performed separately for every Modelica package of the library, which can lead to redundant and duplicated jobs and statements in the CI-structure. These redundancies can be reduced by using a combination of Python to create the GitLab templates and the `extends` command in GitLab-CI/CD. Basic jobs, such as the Modelica check of a package, only have to be defined once as a base job in the Mako templates. Using for-loops, variables and the extends command, when exporting the GitLab templates, the individual jobs are then created for each package and adjusted by variables for the respective package.
Using the artifact functionality, the results and outputs of the various tasks and phases can be made available for download or later publication to provide a more structured view.
The complete CI-structure, if all stages are selected, is shown in Figure 3. The script section of each job is comparably short, as the functionality itself is outsourced into an additional Python library that currently comes with the template generation repository.
Since not all phases should be executed in all scenarios, we use the implementation of `rules` in GitLab-CI/CD to set different triggers for each job. In the current CI-structure, we use `rules` in three ways.

1. Trigger specific jobs based on specific commit messages. E.g. to allow the creation of reference results through a specific commit.
2. Identify commits on special branches like development and main. This allows to handle main and development branches different than feature branches.
3. Identify commits on branches with existing pull request. Same as for special branches.

To exclude specific models from certain stages, we integrated a whitelist functionality. This is useful, if the library author is aware that some already existing models are failing, but this should not impede the development or integration of new models. In addition, the whitelist functionality is necessary if the existing library contains models from other libraries, but does not manage these itself. In this case, errors in the source library models should be detected and fixed within this source library and not lead to failed tests in the extended library. If

[13] https://www.makotemplates.org/

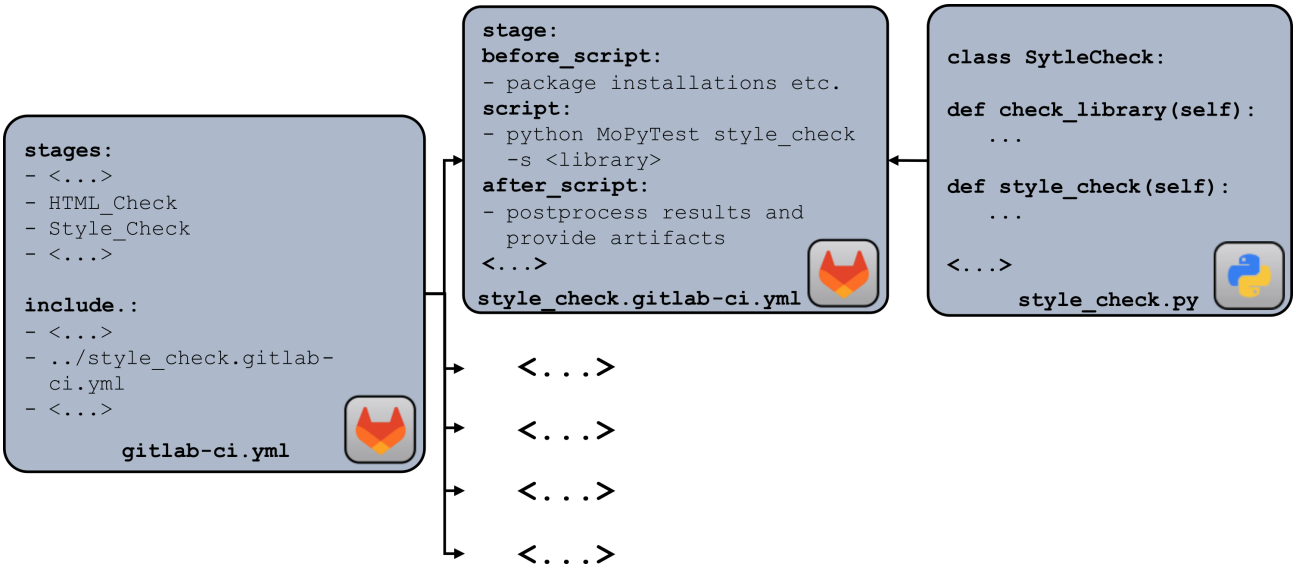

Figure 2. Excerpt of the exported templates (simplified).

DOI
10.3384/ecp204521

these models are not tested, the CI runtime will also be reduced. This is particularly relevant since the IBPSA library represents such a library, on which four libraries are based.
Additionally, an option exists, to allow certain stages to fail. This may be useful if, for example, the library's current status with respect to style checking does not meet the requirements, but insights into style quality are still of interest.

3.4.1 Description of Stages

Figure 3 presents the stages of the created CI-structure. The scattered stages hold jobs which are only executed under special conditions, while the other jobs are executed every time. As some jobs currently require a Dymola installation, we added information about the compatibility with Dymola and OpenModelica to the figure. Following, we give a detailed description of each job and stage, its functionality and its output.
The **Regression Result** stage is conditional and only executed if a pull request is opened for this branch. The executed jobs in this stage will create missingreference results. This stage is beneficial because manually creating reference results requires a Python installation on Linux with BuildingsPy, which raises the hurdle for creating examples with reference results, especially for inexperienced developers. To create the reference results, the developer only needs to add the `.mos` script for the simulation model that holds information about the model to simulate, the experiment settings, and the variables of the models that should be taken into account for regression testing. By comparing the existing `.mos` scripts with existing regression results, the CI identifies not existing regression results and creates them. The resulting reference results are directly pushed to the branch and a new CI pipeline is triggered that is based on the updated reference results.
This automated process only creates reference results that do not yet exist in order to avoid unwanted changes to reference results. However, the process can also be used to update existing reference results. To do this, the developer can delete existing reference results. This way, new updated reference results are generated throughout the explained process. This semi-automatic procedure leaves the sovereignty over the reference results with the developers and yet simplifies the process.
The **Create Whitelists** stage creates whitelists if the used library extends models of another library (e.g. AixLib extending IBPSA). The stage is only executed, if one of the listed commit messages in 3 is used.
Modelica uses HTML code for model documentation. The **HTML-Check** stage checks the HTML code in all Modelica files against valid HTML syntax using the python package PyTidyLib[14] and is always executed. If the check is not successful, due to invalid HTML syntax, a new branch will be created using an API and an automatic repair process is performed on the existing HTML code to fix common errors in the HTML section. Subsequently, the check is performed again and if the check succeeds, a merge request is created on the target Modelica repository with the fixed HTML code. This way there is still a manual review process, but the Modelica

[14] `http://countergram.github.io/pytidylib/`

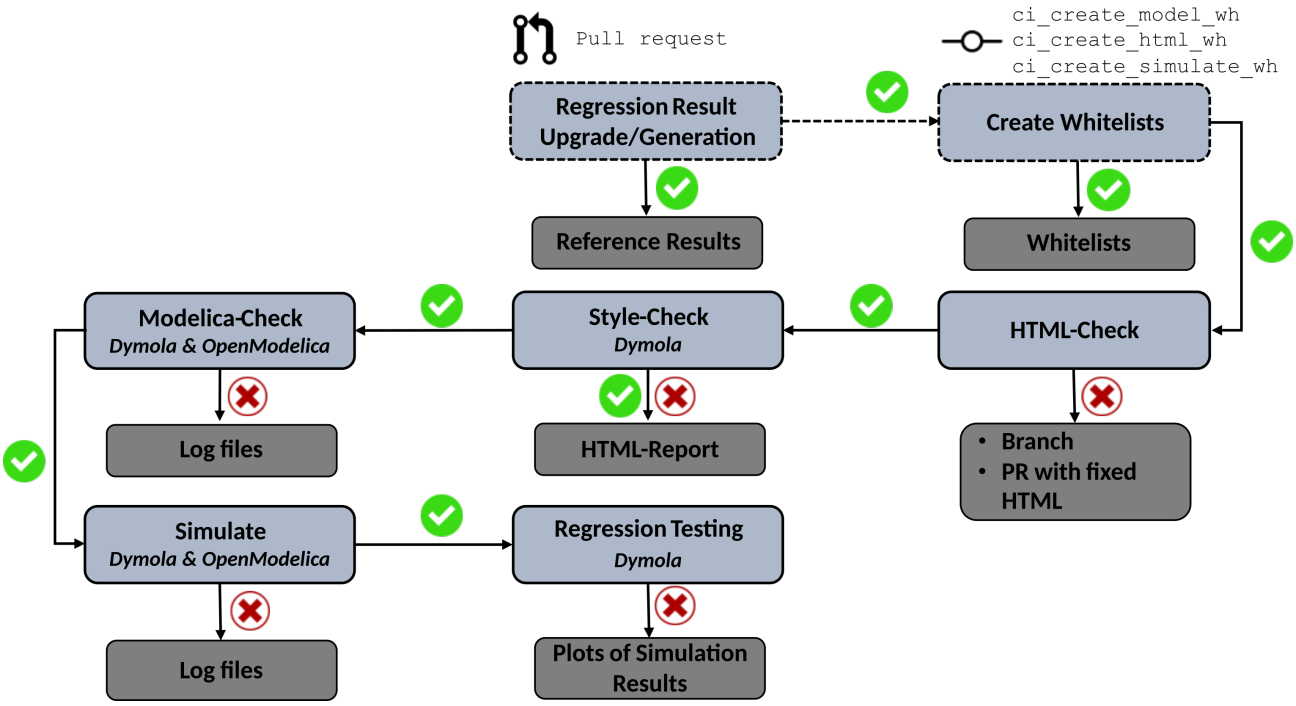

Figure 3. Stages of the invented CI-structure, scattered stages are conditional.

user itself does not have to deal with common issues like unclosed HTML tags.
Next, a **Style-Check** stage is performed using the *Model Management* library of Dymola and runs its inbuilt style check against the whole library. The checking process includes three types of checks: class checks, component checks and general checks. These checks among other things evaluate if the existing model code holds correct documentation, descriptions and class names, but does not perform any checks regarding model functionality. As the *Model Management* can only be used by Dymola this stage is currently limited to the usage of Dymola. The result of the stage is a HTML-report about the quality of the checked library that gives detailed insights about the quality. The report is stored as an `artifact` which is available for download.
Subsequently, the **Modelica-Check** gives detailed insights about the syntactic and logical correctness of the model code by using the check functionality of Dymola or OpenModelica. If errors arise, these will be saved to a log file, which is available via artifacts. The stage is implemented for both, OpenModelica and Dymola.
The **Simulate** stage runs all models inside the library which extend the *Modelica.Icons.Example* model. This is useful, because a model might pass the previous checks, but won't simulate successful. The stage is available for OpenModelica and Dymola.
Regression testing stage runs simulation for all models against their existing regression results using the BuildingsPy.
In case of a failure in the regression tests, plots of the simulation results are created. This stage should allow a fast identification of which model failed and why by creating plots of the expected and the new result trajectory using Google Charts[15]. The plots will be deployed directly via an GitLab page and be posted to the GitHub pull request, see Section 3.4.3
As some jobs, like simulation of the models or regression testing, are computationally and time intensive, we implemented the option to only run these jobs for models, where the source code of the model changed throughout a commit. By using the `git diff` feature, we can check the differences between the target branch and the current branch and identify the changed models. This function currently does not consider inheritance and integration of submodels in other models. This means that if a single component that is part of multiple models changes, only the component itself is checked, but not all models in which it is integrated. For certain events, like the assignment of a reviewer in a pull request, this option is disabled and all models, even if not changed, are checked.

[15] `https://developers.google.com/chart`

3.4.2 *IBPSA* Library Specific

In addition to the stages shown and described, there is also the **IBPSA-Merge**. This is very tailored to the existing setup of IBPSA library and extending libraries and therefore not shown in the general process schema. This stage allows performing an automatic merge of the source library *IBPSA* into the extending libraries like *AixLib*. Therefore, the merge script delivered by *BuildingsPy* is used. After the automatic merge, a conversion script is created based on the existing conversion script of *IBPSA* and the latest conversion script of the extending library.
Additionally, we add the annotation `__Dymola_LockedEditing` to all IBPSA models, which allows displaying these models as locked inside Dymola as shown in Figure 4. This is very useful to prevent changes to Modelica models that are not part of the extending library, which would lead to merge conflicts when performing the next *IBPSA* merge.

3.4.3 Interfacing and Communication

Automating tasks like the *IBPSA* merge, or the creation of reference results, can save a lot of time when maintaining a Modelica library. But not all tasks can be completely automated and even if a complete automation is possible, the results need to be communicated with the library users. To fulfill the need for communication, we use the GitHub REST API. If using GitHub as the main repository, the GitHub REST API allows writing messages via a bot account which give feedback and instructions. E.g. in case of failed regression tests, the GitLab page with plots of the simulation results will be linked to the corresponding pull request, so that the user can directly see which models are failing and might also be able to identify why the simulation results differ from the regression results.

4 Exemplary Application on Two Libraries: *AixLib* and *BESMod*

In the following sections, we provide insights into how we implemented a working CI infrastructure at our institute (4.1) and show the application of the developed CI infrastructure to two libraries: *AixLib* (4.2) and *BESMod* (4.3).

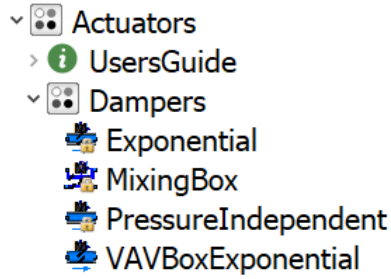

Figure 4. Locked *IBPSA* components in *AixLib* library.

DOI
10.3384/ecp204521

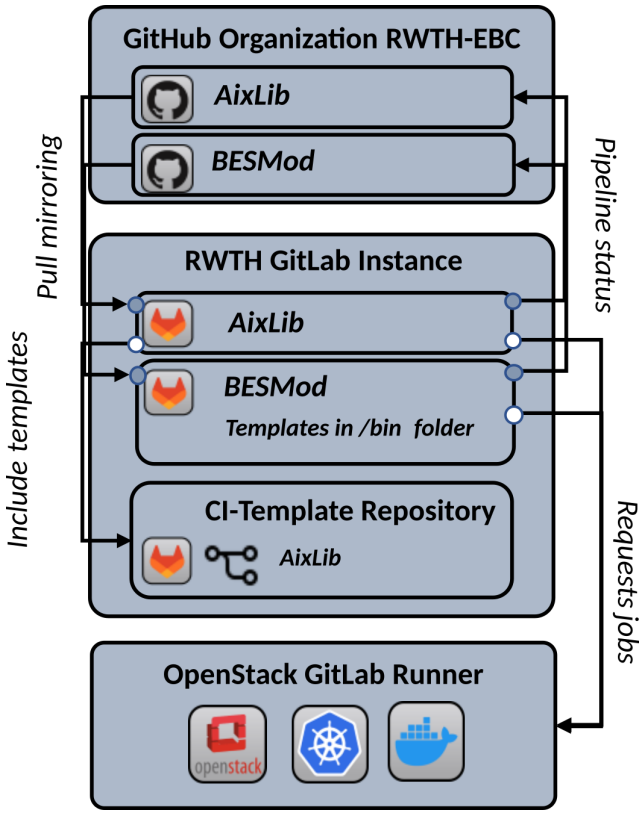

Figure 5. CI-infrastructure at RWTH in Aachen.

4.1 Setup at RWTH in Aachen

The used setup at our institute in Aachen is shown in Figure 5. We have a public available GitHub organization that hosts both later described libraries *AixLib* and *BESMod*. Both libraries are mirrored to our university GitLab instance, where we have a specific subgroup for mirrored projects. The jobs are executed by a scalable GitLab runner, provided through the GitLab runner docker image and Kubernetes. The Runner setup is hosted on our internal cloud service OpenStack. Both *AixLib* and *BESMod* hold a `gitlab.ci-yml` file. In case of *AixLib* this links to another GitLab repository, where the template structure, created by *MoCITempGen* is stored in a separated branch. This way, we can easily implement changes to the CI, by simply updating the external repository branch of the templates. For *BESMod* the created templates are stored directly in the repository in the `bin` folder.

4.2 *AixLib*

The development of *MoCITempGen* took place on the basis of *AixLib* (Maier et al. 2023). A brief explanation of the application of *MoCITempGen* at *AixLib* has already been given in this previous paper. The template generator was developed based on the existing CI of *AixLib* and generalized so that it can be applied to other Modelica libraries.

To show that the concept of *MoCITempGen* works, we opened a demonstration pull request[16] in the *AixLib* and provoked the CI to perform to give two examples of how the CI works and interacts with the modeler.

The second example is a failing regression test, that was provoked by changing the scaling factor for the heat pump model in the example `AixLib.Fluid.HeatPumps.Examples.-HeatPump`. Due to the changes to the model, the regression test stage fails and the *ebc-aixlib-bot* posts a comment into the pull request and links the GitLab page with the plots showing the differences between existing regression results and new results. A screenshot is shown in Figure 6a. The second example is the creation of new reference results. Therefore, we deleted the existing results and pushed them to the branch. The CI notices a missing reference result for an existing simulate and plot `.mos` script and creates new results, pushes them to the branch, and informs about the new created results inside the pull request with a link to the GitLab page with plots of the new reference results (see Figure 6b). Further information about the usage of *AixLibs* CI can be found in the *AixLib*-Wiki[17].

4.3 *BESMod*

Contrary to the *AixLib*, the library *BESMod* is not an extension to the library *IBPSA*. Rather, it uses currently existing libraries, such as the *IBPSA*, *Buildings* or *AixLib*. Thus, to load the *BESMod* in the CI, installation of these additional requirements is necessary. GitLab-CI offers *before-scripts* to execute specific commands prior to the actual script. Before generating the CI configurations using *MoCITempGen*, adding an additional line for requirement installation to the *before-script* section in the *.txt* template files was required. Afterwards, all features of the CI were directly accessible and useable, even for a more complicated setup, as is the case in *BESMod*. In summary, while smaller adjustments may be necessary to a specific library, *MoCITempGen* decreases the CI setup time drastically.

5 Discussion

The presented methodology was successfully applied to two Modelica libraries. Even though the template creation tool was developed with the goal of high flexibility, there are currently still some requirements. These requirements are:

1. the target Modelica library must be hosted in or mirrored to a GitLab repository,
2. a GitLab runner must be configured (via SaaS or by hosting an own),
3. for jobs that need a simulation environment, either an OpenModelica or Dymola image must be provided
4. in case of using Dymola, a Dymola license is required

The limitation to GitLab repositories can be circumvented by using the GitLab mirroring function in section 3 which

[16] `https://github.com/RWTH-EBC/AixLib/pull/1389`

[17] `https://github.com/RWTH-EBC/AixLib/wiki/GitLab-CI`

(a) New and existing results for failed regression test.

(b) New regression results created by CI process.

Figure 6. Example of AixLib CI for regression testing.

allows applying the presented approach to GitHub, BitBucket or AWS CodeCommit. The required GitLab runner can be self-hosted without requiring to pay any service as described in section 2. However, the presented version of MoCITempGen currently enforces the use of GitLab-CI/CD in the background, which requires familiarization with the GitLab-CI/CD syntax and runner infrastructure. Furthermore, as described in section 3, some stages need Dymola, or at least OpenModelica. Dymola requires a paid license. OpenModelica on the other hand is open source and already offers public available images on DockerHub[18]. But currently, not all stages are compatible with OpenModelica. Therefore, we are aiming to make all stages available via OpenModelica in the future. Additionally, there are further possibilities for improvement. E.g., the outputs of the different stages are not uniform. Some stages output log files, others HTML-reports, others files to download. This could be unified with a central report, which holds all relevant information.

6 Conclusion and Outlook

This paper gives an overview of CI applications in the context of Modelica and presents the tool *MoCITempGen* that aims to facilitate the use of CI for authors of Modelica libraries. The tool and the underlying methodology are explained, and the application of the tool on two Modelica libraries is shown. Even though we have applied *MoCITempGen* to two libraries in the context of the building energy efficiency sector, it is also possible to apply it to other libraries from other domains.

In order to increase the application possibilities and to support different repository architectures, we want to extend the tool in the future. This concerns on the one hand the support of OpenModelica in all stages, in which a simulation environment is used. On the other hand, the export format of the templates will be extended so that templates for GitHub Actions can also be exported in the future. This is especially useful since GitHub Actions also supports the possibility of self-hosted runners.

Furthermore, we want to increase the flexibility and maintainability by separating the Python library from the template generation tool.

Eventually, the goal is to unify the various existing approaches to CI in the context of Modelica and make them available across use cases.

[18] `https://hub.docker.com/r/openmodelica/openmodelica/tags`

Acknowledgements

This work has been supported by the German Federal Ministry for Economic Affairs and Climate Action as part of the project BIM2Praxis (grant numbers 3EN1050A, 3EN1050B).

References

Beck, Kent (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional. ISBN: 978-0-201-61641-5.

Booch, Grady (1991). *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company. ISBN: 978-0-8053-0091-8.

Gall, Leo et al. (2021-09-27). "Continuous Development and Management of Credible Modelica Models". In: *Modelica Conferences*, pp. 359–372. ISSN: 1650-3740. DOI: 10.3384/ecp21181359.

Hugues, Jerome et al. (2022-03-24). *TwinOps: Digital Twins Meets DevOps*. report. Carnegie Mellon University. DOI: 10.1184/R1/19184915.v2.

Maier, Laura et al. (2023). "AixLib: an open-source Modelica library for compound building energy systems from component to district level with automated quality management". In: *Journal of Building Performance Simulation* 0.0, pp. 1–24. DOI: 10.1080/19401493.2023.2250521. eprint: https://doi.org/10.1080/19401493.2023.2250521. URL: https://doi.org/10.1080/19401493.2023.2250521.

Rabuzin, Tin, Maxime Baudette, and Luigi Vanfretti (2017-07-01). *Implementation of a continuous integration workflow for a power system Modelica library*. Pages: 5. 1 p. DOI: 10.1109/PESGM.2017.8274618.

Schölzel, Christopher et al. (2021-07-19). "Countering reproducibility issues in mathematical models with software engineering techniques: A case study using a one-dimensional mathematical model of the atrioventricular node". In: *PLoS ONE* 16.7, e0254749. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0254749.

Vasilescu, Bogdan et al. (2015-08-30). "Quality and productivity outcomes relating to continuous integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, pp. 805–816. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786850.

Wetter, Michael (2019). *BuildingsPy*. Language: en. DOI: 10.11578/DC.20190430.2.

Wetter, Michael et al. (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.