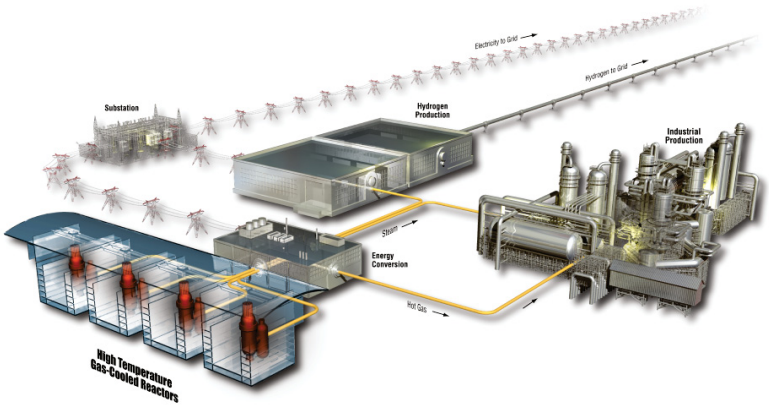
INL/EXT-16-40004
Revision 0

Software Development Infrastructure for the HYBRID Modeling and Simulation Project

Aaron S. Epiney
Robert A. Kinoshita
Jong Suk Kim
Cristian Rabiti
M. Scott Greenwood (ORNL)

September 2016

The INL is a
U.S. Department of Energy
National Laboratory
operated by
Battelle Energy Alliance



Idaho National Laboratory

INL/EXT-16-40004

HYBRID Modeling and Simulation

Software development infrastructure for the HYBRID modeling and simulation project

Aaron S. Epiney
Robert A. Kinoshita
Jong Suk Kim
Cristian Rabiti
M. Scott Greenwood (ORNL)

September 2016

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

EXECUTIVE SUMMARY

One of the goals of the HYBRID modeling and simulation project is to assess the economic viability of hybrid systems in a market that contains renewable energy sources such as wind and solar. The overarching concept is that it is possible for the nuclear plant in a nuclear-renewable hybrid energy system (NHES) to sell products and commodities in addition to electricity. These alternative commodities absorb (at least partially) the volatility introduced on the grid by the variable renewable energy sources. Candidate systems are currently modeled in the Modelica programming language. To assess system economics, an optimization procedure is used to find the minimal cost for electricity production. The RAVEN code is used as a driver for the problem.

The modeling and simulation project involves teams from three national laboratories, namely Argonne National Laboratory (ANL), Oak Ridge National Laboratory (ORNL), and Idaho National Laboratory (INL), that develop the different Modelica component models as well as the RAVEN driver and optimization framework. Computer infrastructure and development procedures have been established to simplify and organize the development across the three laboratories.

The core of the software development infrastructure is the project management software GitLab. This software is a web-based Git repository manager with wiki and issue tracking features. The Git repository for the HYBRID modeling and simulation project is hosted on the INL High Performance Computing (HPC) cluster. The repository is accessible at the following url: git@hpcgitlab.inl.gov:hybrid/hybrid.git. Access to the repository is controlled and every new collaborator to the project must request access.

A repository structure has been established that defines where the developers have to store models and test cases. The NHES directory hierarchy is derived from the existing standard format used for the Modelica standard library. The implementation of the actual nuclear hybrid energy system is performed within the "Systems" package that is organized to allow compartmentalized subsystem development, which enables clear boundaries of each system.

It is assumed at this stage that the HYBRID modeling and simulation framework can be classified as non-safety "research and development" software. Hence, the associated quality level is Quality Level 3 (QL3) software. This imposes low requirements on quality control, testing, and documentation early in the software development process. The quality level could change as the application development continues.

Despite the QL3 requirement level, a workflow for the HYBRID developers has been defined that includes a coding standard and some documentation and testing requirements. The repository performs automated unit testing of contributed models. The automated testing is achieved via an open-source python script called BuildingsPy from Lawrence Berkeley National Lab. BuildingsPy runs Modelica simulation tests using Dymola in an automated manner and generates and runs unit tests from Modelica scripts written by developers.

In order to assure effective communication between the different national laboratories a biweekly videoconference has been set-up, where developers can report their progress and issues. In addition, periodic face-to-face meetings are organized to discuss high-level strategy decisions. A second means of communication is the developer email list. This is a list to which collaborators can send emails that will be received by the collective of the developers and managers involved in the project. Third, to exchange documents quickly, a SharePoint directory has been established. SharePoint allows teams and organizations to intelligently share and collaborate on content from any location.

CONTENTS

FIGURES

ACRONYMS

ANL	Argonne National Laboratory
CCB	Change Control Board
DEAR	Department of Energy Acquisition Regulation
DOE	Department of Energy
Dymola	Dynamic Modeling Laboratory
FMU	Functional Mock-up Unit
FN	Foreign National
GIF	Generation IV International Forum
Git	Global Information Tracker
GitLab	Git Laboratory
GUI	Graphical User Interface
HPC	High Performance Computing
HVAC	Heating, Ventilation, and Air Conditioning
INL	Idaho National Laboratory
IRIS	International Reactor Innovative and Secure
ISMS	Integrated Safety Management System
LBL	Lawrence Berkeley Laboratory
MOOSE	Multiphysics Object-Oriented Simulation Environment
MSL	Modelica Standard Library
NHES	Nuclear Hybrid Energy System
ORNL	Oak Ridge National Laboratory
PHS	Primary Heat System
POC	Point of Contact
QA	Quality Assurance
QL3	Quality Level 3
RAVEN	Risk Analysis Virtual Environment
SVN	Apache Subversion

1. Introduction

One of the goals of the HYBRID modeling and simulation project is to assess the economic viability of hybrid systems in a market that contains renewable energy sources such as wind and solar [1]. A candidate nuclear-renewable hybrid energy system (NHES) would include a nuclear reactor that generates electricity for the grid and provides heat and/or electricity to another plant that produces a valuable product, such as hydrogen or potable water. The overarching hypothesis is that sale of non-electric energy commodities can dampen (at least partially) the volatility introduced by variable renewable energy sources [2].

The project assumes a modular energy system that is comprised of an assembly of components. For example, a system may include a nuclear reactor, a gas turbine, a battery, and some renewables. Candidate systems are currently modeled in the Modelica [3,4] programming language, but other languages and codes may be envisaged in the future.

To assess the system economics, an optimization procedure is applied to vary different system parameters to find the minimum cost of electricity production. Figure 1 shows a diagram of the software framework for the HYBRID modeling and optimization. As one can see, the statistics and optimization code Risk Analysis Virtual Environment (RAVEN) [5] is used as a driver for the whole problem. RAVEN runs the optimization by varying selected parameters (i.e., changing input parameters in the system model), running the Modelica system model, collecting output from Modelica, and determining the next optimization step. FMU blocks in Figure 1 are Functional Mock-up Units in the Modelica terminology.

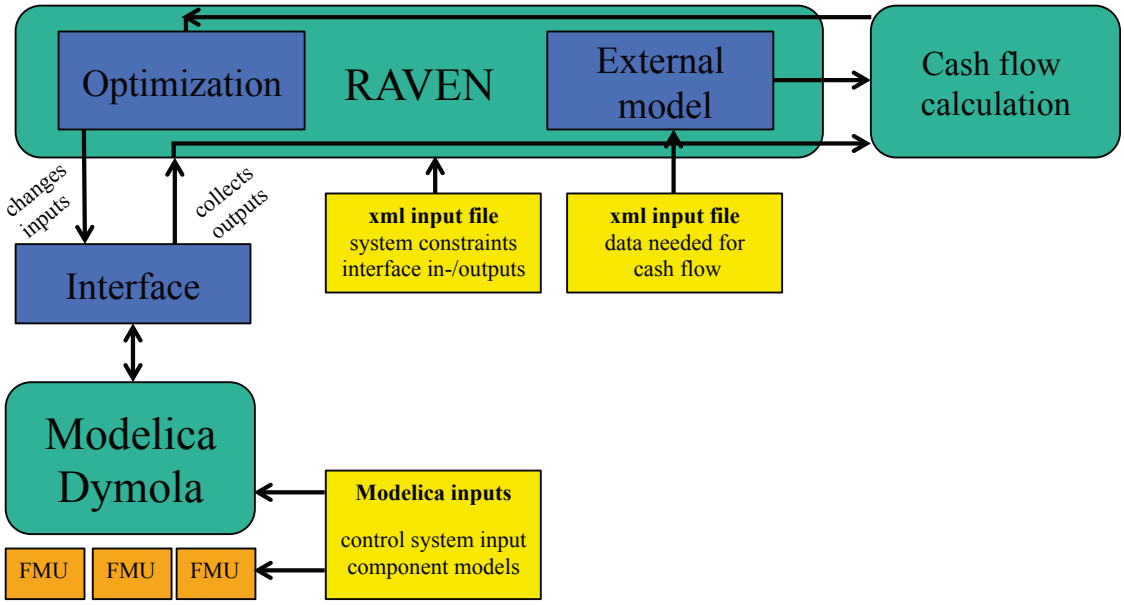


Figure 1. HYBRID system modeling and optimization framework.

Research groups in different national laboratories develop the various Modelica component models and the RAVEN driver and optimization framework. The modeling and simulation project currently involves teams from three national laboratories, namely Argonne National Laboratory (ANL), Oak Ridge National Laboratory (ORNL) and Idaho National Laboratory (INL). The project includes heavy software framework and model development.

Computer infrastructure and development procedures have been established to simplify and organize the software framework development across the different laboratories. In addition, some documentation requirements, coding standards, and file sharing practices have been defined. Common procedures add structure to the workflow and establish the foundation that will be necessary for future increased quality assurance requirements. Strict software QA may be required in the future as the project scope and model and software complexity grows. Therefore, all the tool and procedures installed and described in this document are compliant for software development under the 'Research and Development' categorization.

The purpose of this document is to introduce and describe the infrastructure and software development tools used in the HYBRID project. Many of these tools and procedures have been borrowed from other ongoing software development activities at INL, such as Multiphysics Object-Oriented Simulation Environment (MOOSE) and RAVEN. The current document describes how these requirements have been adjusted to meet the needs of the HYBRID modeling and simulation project.

2. GitLab Repository

The core of the software development infrastructure for the HYBRID modeling and simulation project is the project management software GitLab. The software is a web-based Git repository manager with wiki and issue tracking features. GitLab is the officially supported project management tool for software development at INL. The high performance computing (HPC) division at INL hosts and supports a GitLab server for applications developed at INL.

One of the most common project management software tool types is scheduling tools. Scheduling tools are used to sequence project activities and assign dates and resources to them. The detail and sophistication of a schedule produced by a scheduling tool can vary considerably with the project management methodology used and the scheduling methods supported [6]. GitLab supports, among others:

- Multiple dependency relationship types between activities
- Resource assignment and leveling
- Critical path
- Activity duration estimation and probability-based simulation

Project planning software can be expected to provide information to various people or stakeholders, and can be used to measure and justify the level of effort required to complete the project. These requirements might include:

- Overview information on how long tasks will take to complete
- Early warning of any risks to the project
- Information on workload (e.g., to include planning for holidays)
- Evidence of work performed
- Historical information on how projects have progressed; in particular, planned versus actual performance
- Optimum utilization of available resource
- Collaboration with each teammates and customers
- Instant communication to collaborators

This section of the report explains how GitLab has been set up, how access to the repository is managed, and how the repository is used in the context of the HYBRID modeling and simulation project.

2.1 Git Repository Manager

The first part of GitLab is the Git repository manager. Git is a version control system that is used for software development [7] and other version control tasks. As a distributed revision control system it is aimed at speed, data integrity, and support for distributed, non-linear workflows. Linus Torvalds created Git in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.

Revision control manages changes to a set of data over time [8]. These changes can be structured in various ways. Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This approach can cause problems when file identity changes, such as during renaming, splitting, or merging of files. Git instead considers changes to the data as a whole. This latter approach is less intuitive for simple changes, but simplifies more complex changes.

As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server [9]. This makes it nearly impossible to lose any data since every user has a full back-up of the repository. Git is free software distributed under the terms of the GNU General Public License version 2.

Version management with Git makes branching and merging much easier than older versioning systems such as Apache Subversion SVN. This allows a wide variety of branching strategies and workflows. Almost all of these are an improvement over the methods used before Git. To use the Git repository in an efficient manner, the workflow must be clearly defined and simple. Furthermore, the workflow should be integrated with the issue tracking system. Figure 2 shows an example of a possible evolution of the Git repository. It shows feature driven development and feature branches with issue tracking.

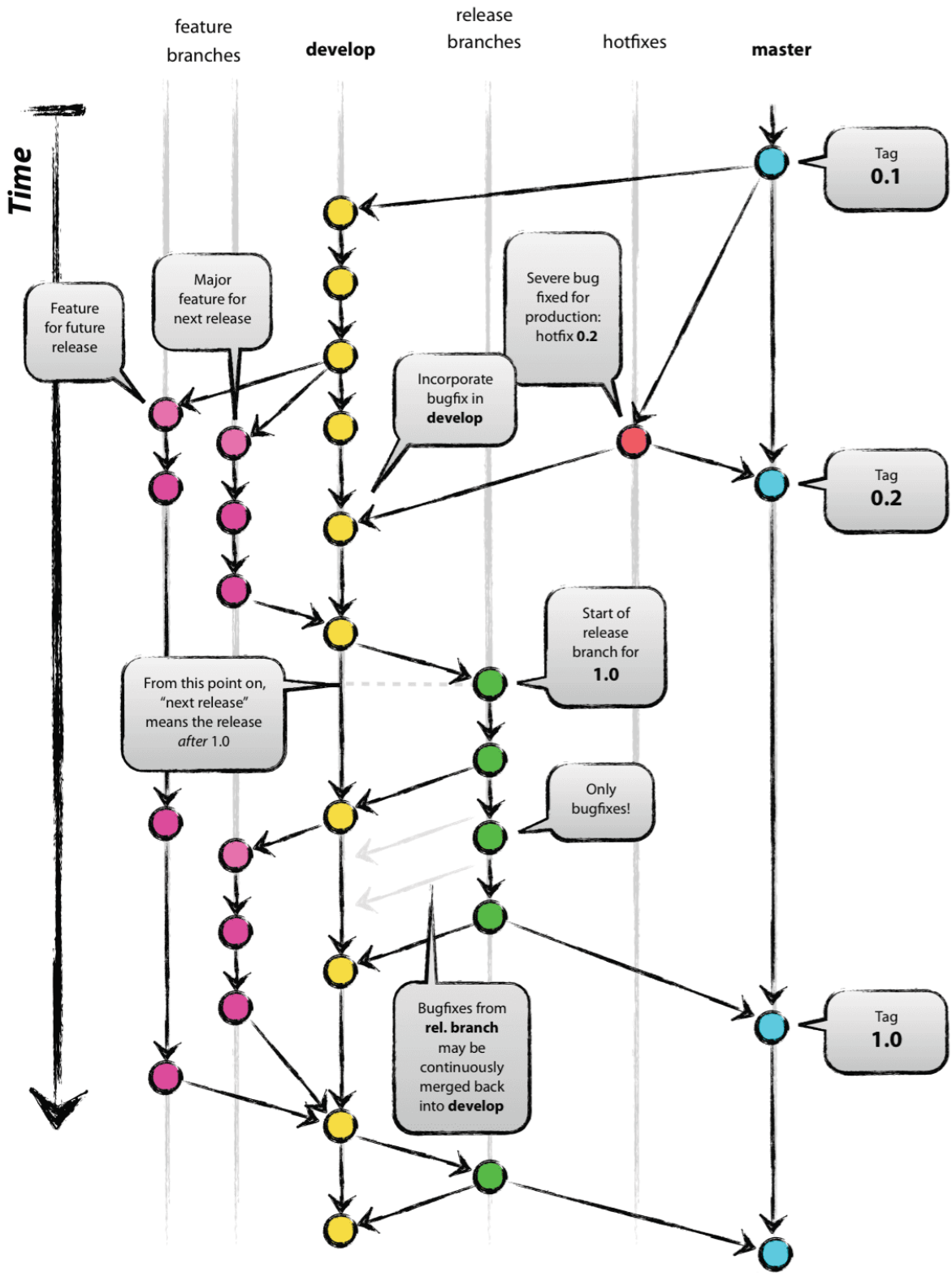


Figure 2. Example timeline for a Git repository [10].

The Git repository for the HYBRID modeling and simulation project is hosted on the INL HPC cluster. It is accessible under the following url:

git@hpcgitlab.inl.gov:hybrid/hybrid.git

2.2 Wiki and Issue Tracking System

In addition to the Git repository, GitLab provides a graphical user interface (GUI) that can be viewed from within any web browser. The GUI allows browsing and viewing the source code and other files checked in to the repository, and it also provides an issue tracking system and a wiki. Figure 3 shows the project page that is displayed to users after login to the HYBRID modeling and simulation GitLab page.

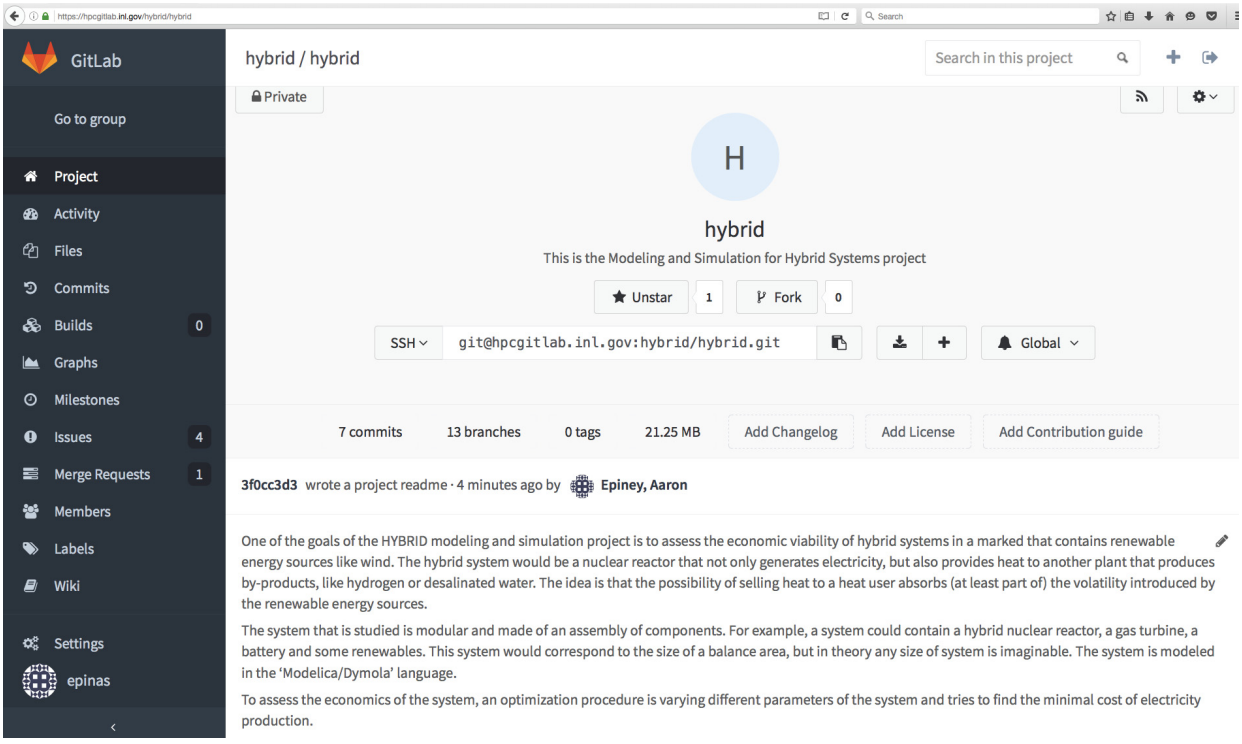


Figure 3. HYBRID modeling and simulation GitLab project page.

2.2.1 Issue Tracking System

The issue tracking system, also called ticketing system, is a computer software package that manages and maintains lists of issues, as needed by an organization. Each 'issue' creates a new 'ticket'. An issue can be anything from a defect to a new required capability in the software.

New issues can be opened by any registered user. This provides external users of the software a direct way for communicating desired updates or encountered problems to the development team. Consistent use of an issue or bug tracking system is considered one of the "hallmarks of a good software team" [11].

A ticket element, within an issue tracking system, is a running report on a particular problem or development, its status, and other relevant data. It also includes the log of all communication between developers and users concerning the particular issue. Each ticket has a unique reference number, also known as a case or issue number, which is used to allow the user or developers to quickly locate, add to or communicate the status of the issue. Issues can have several aspects to them. Each issue in the system may have an urgency value assigned to it, based on the overall importance of that issue. Other details of issues include date of submission, a detailed description, solutions, or work-arounds, and other relevant information. As previously noted, each issue maintains a history of each change. Figure 4 is an example of a ticket in GitLab.

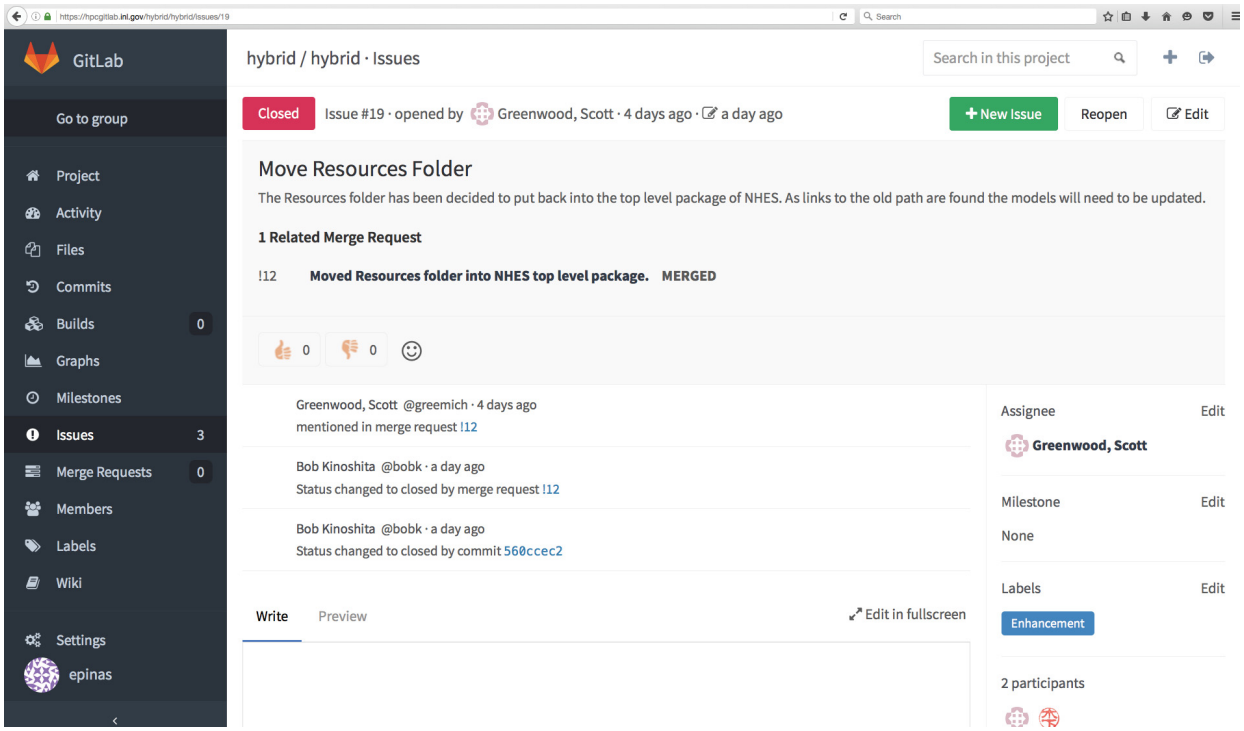


Figure 4. HYBRID modeling and simulation GitLab example ticket.

2.2.2 Wiki

In addition to the Git repository and the issue tracking system, the GitLab software package also includes a wiki. A wiki is a website that provides collaborative modification of its content and structure directly from the web browser. The wiki allows the users and developers to create web pages to distribute information that is not part of a ticket, like user manuals, installation instructions, quality assurance documents, etc. Every user or developer can create a new page

and link it with other pages or edit existing pages. A simple mark-up language is used to create the webpages. The so-called "wiki-markup" language allows writers do define sections, subsections, and lists, highlight text or source code blocks, etc.

The wiki for the HYBRID project currently contains pages that describe the installation process of the HYBRID repository, including how to get RAVEN as a submodule of the HYBRID repository and instructions on how to access the repository from outside INL. Other pages describe the Modelica environment description and the development workflow (see Figure 5). This is an initial set of pages and, as described above, every user and developer can and is encouraged to create more new pages if needed.

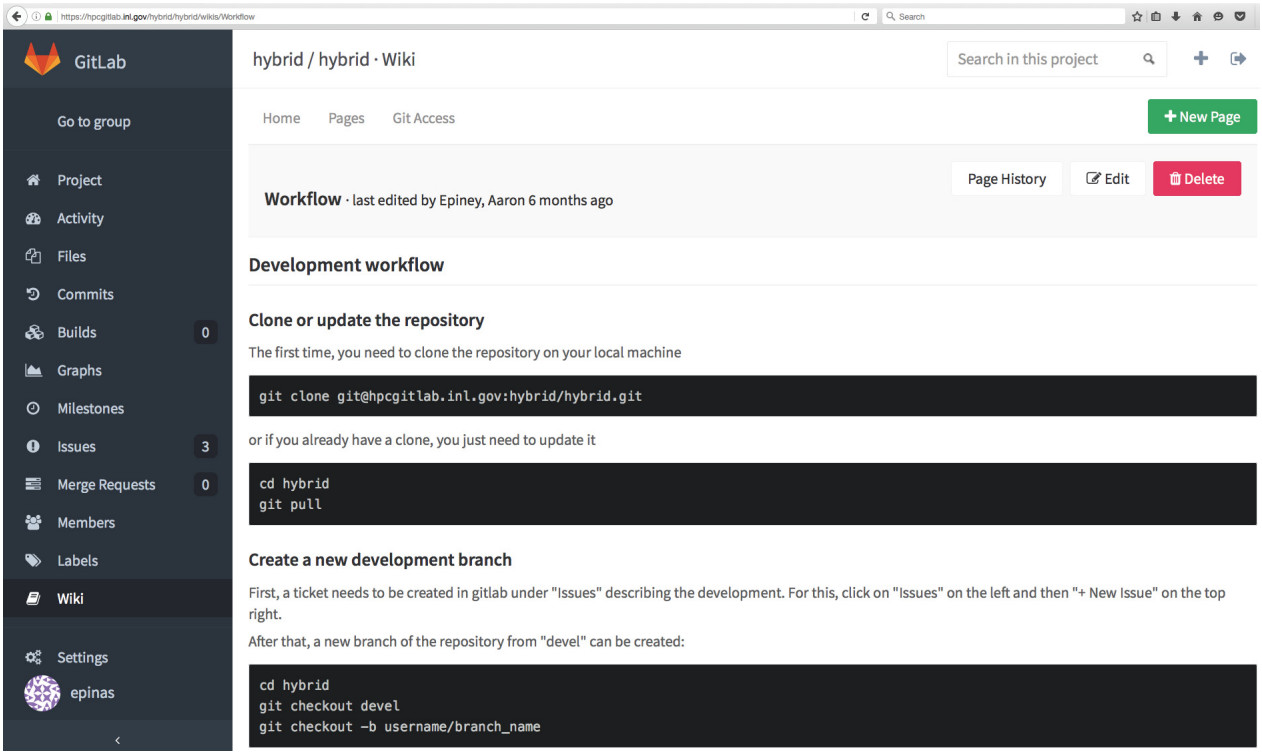


Figure 5. Example wiki page for HYBRID modeling and simulation.

2.3 GitLab Repository Access

GitLab, including the Git repository, ticketing system, and wiki, are hosted on INL's HCP servers. Every new collaborator to the HYBRID modeling and simulation project must request access to the repository. The process is identical for people internal or external to INL. Figure 6 shows schematically how access is managed. First, the repository manager determines if the new user already has access to INL's HPC cluster. If not, the user must first request an HPC account by filling in an online form. If the user is a U.S. citizen, HPC access is usually granted within a couple of days. The new user is informed by the HPC support that a new user name has been created and issues a password for the user.

If the user is a foreign national ("FN" in Figure 6), a "security plan" must be issued. The security plan includes the user's name and nationality, provides a justification as to why the foreign national needs access to the INL HPC, and specifically adds HYBRID to the plan. After submission, the security plan is checked by different organizations at INL, including export control and cyber security. If the user is not approved for access to either the HPC or the HYBRID repository, he/she cannot participate in the project.

Once granted HPC assess and provided with a user name, the user will then communicate the username to the HYBRID repository administrator that will be able to open access to the repository for the new user. The new user is then capable of checking out the repository and making comments using the GitLab ticketing system and wiki. The repository administrator also adds the new user to the HYBRID developer mailing list.

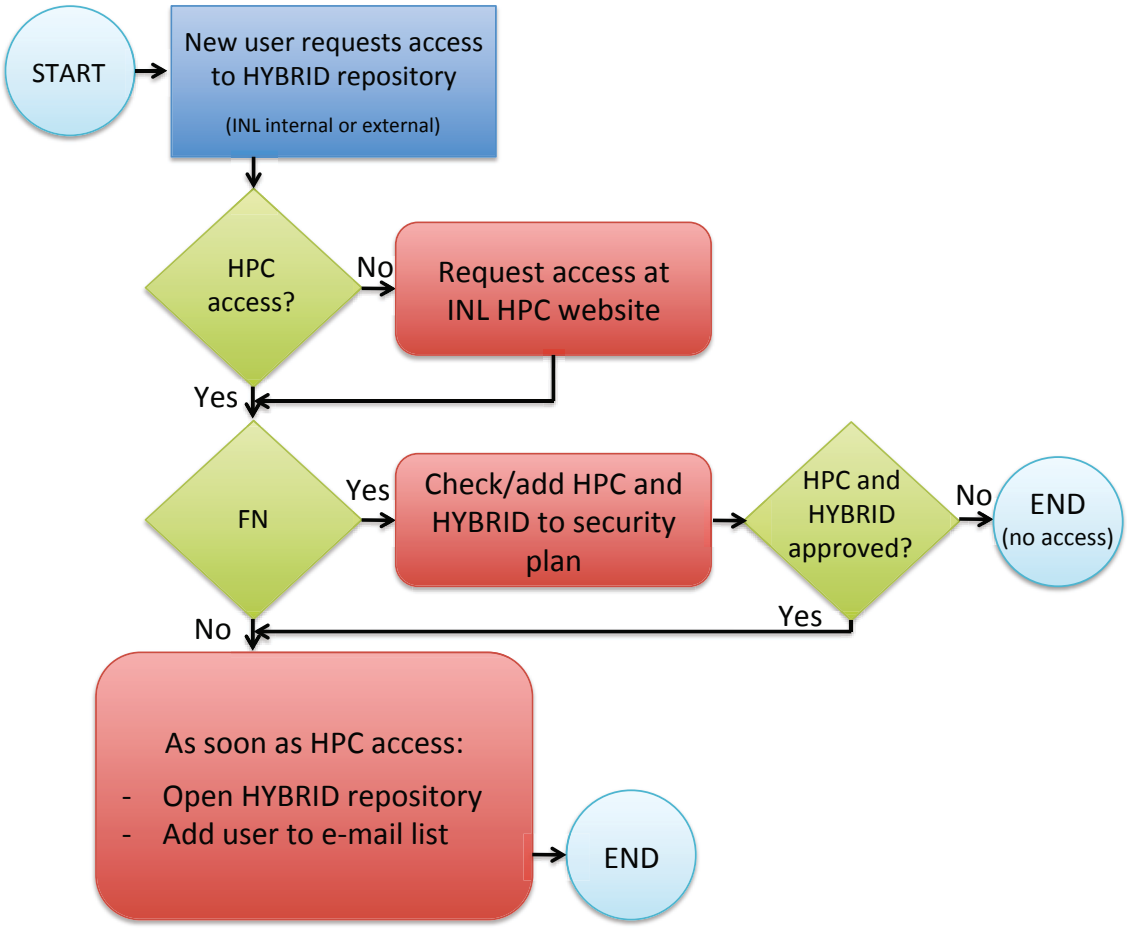


Figure 6. HYBRID access request workflow.

In addition to the developments in the HYBRID repository, the RAVEN code is also used in the project. RAVEN is developed independently from the HYBRID project and, thus, uses its own GitLab repository for project management and version tracking. The RAVEN code Git repository has been made available in the HYBRID repository as a submodule. That means that when checking out the HYBRID repository, the RAVEN code repository can also be checked

out. However, the RAVEN repository has its own access management. The HYBRID user is only able to get the RAVEN code if he is granted access to RAVEN.

Figure 7 shows a flow chart on how to get access to RAVEN when the user already has access to the HYBRID repository. First, the repository manager determines if the HYBRID user already has RAVEN access. If he does, there is nothing to be done and the user can check out RAVEN as a submodule for HYBRID. If the user does not have RAVEN access, the next step is to determine if the institution he is associated with already has an institute license for RAVEN. If not, the HYBRID user's institution must first obtain a RAVEN license. This process is not described here, but information on how to get a RAVEN license for an institute can be obtained from the RAVEN development team [12]. Once the institution has a RAVEN license, the HYBRID user has to be registered as a RAVEN user by his institution. This is straightforward if the HYBRID user is a U.S. citizen. The user just contacts the institution's "RAVEN point of contact (POC)" (the POC for the institution is specified on the RAVEN license agreement) requesting to be a RAVEN user under the institution's license agreement. The POC then communicates to the RAVEN development team that his institution has a new RAVEN user. If the HYBRID user is a foreign national, he must first add RAVEN to his security plan at INL. As for the HPC and HYBRID access, a justification to use RAVEN has to be submitted to the reviewers. If the user is approved for RAVEN, he can proceed by contacting his POC.

Once the POC of the institution communicates the new RAVEN user to INL and the security plan is approved (for foreign nationals only), the RAVEN administrator will open access to the RAVEN repository for the user. He will also add the user to the RAVEN user mailing list. As soon as access to the RAVEN code is granted, the user is able to check out RAVEN as a submodule for HYBRID.

2.4 Repository Structure

A Git repository has been created on hpcgitlab.inl.gov called "hybrid". It will be organized with a folder structure as shown in Figure 8. The purpose of each folder in the 'hybrid' folder will then be described.

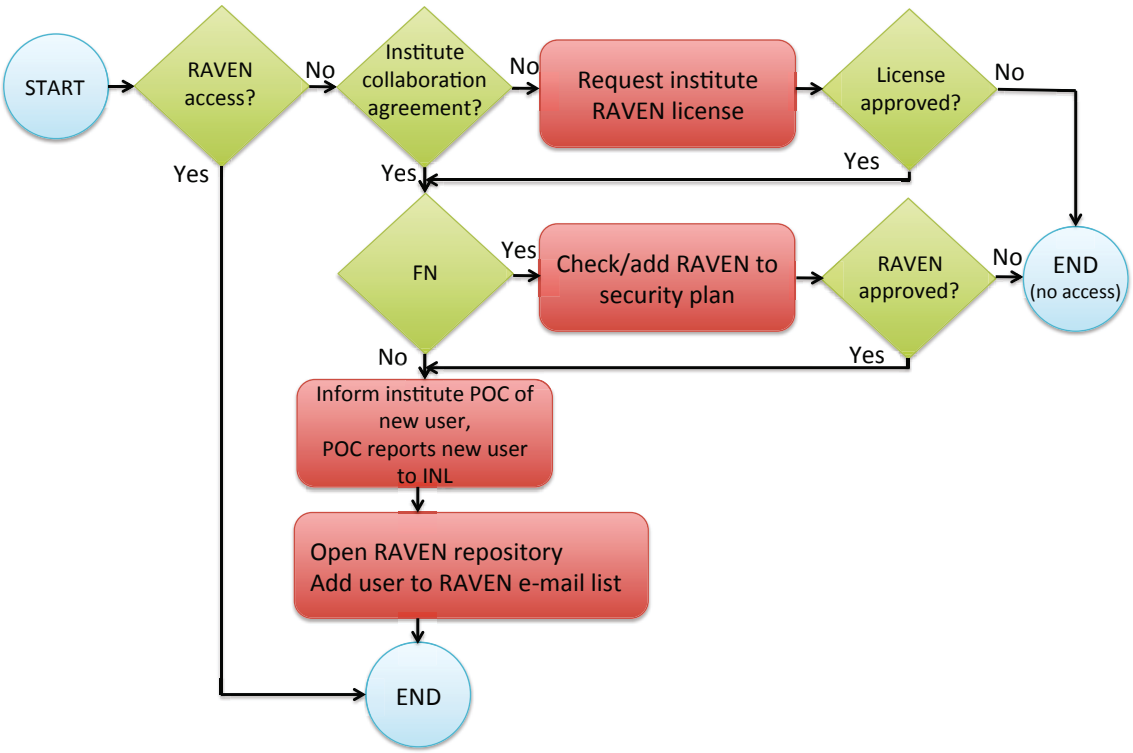


Figure 7. RAVEN as submodule for HYBRID access request workflow.

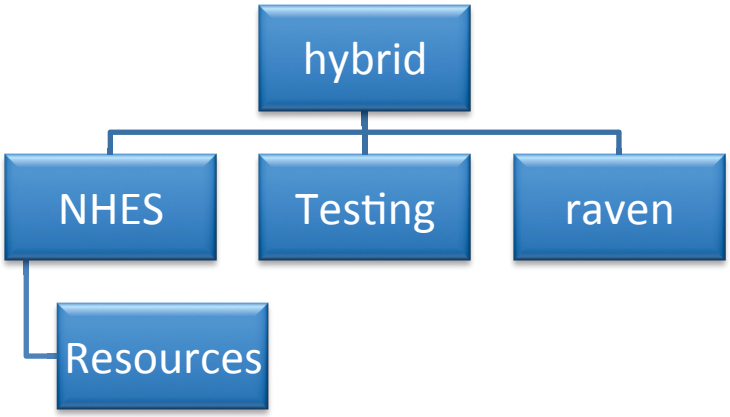


Figure 8. HYBRID repository structure.

2.4.1 NHES

This folder contains component models contributed by project collaborators, written in Modelica. They are arranged as a single Modelica *package*, which is a hierarchical arrangement of code files within nested directories in NHES. This package contains all Modelica code needed to execute project models with the exception of libraries, such as the Modelica Standard Library (MSL) [13] and the open source ThermoPower library [14]. The project team has agreed to standardize on the latest release of ThermoPower, which is version 3.1 beta.0 at the time of this writing. The current library and compiler versions used are reported in the HYBRID wiki.

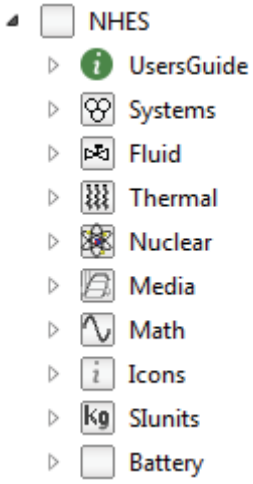


Figure 9. NHES folder hierarchy.

The NHES hierarchy (shown in Figure 9 as it appears in the Dymola simulation environment) is derived from the existing standard format used for the MSL. The category structure of the MSL is further discussed in section 6.4.

Implementation of the actual nuclear hybrid energy system is performed within the "Systems" package. The Systems package is organized to allow compartmentalized subsystem development, which enables one to establish clear boundaries for each subsystem. For example, the base NHES case is a tightly coupled system with the primary subsystems shown in Figure 10.

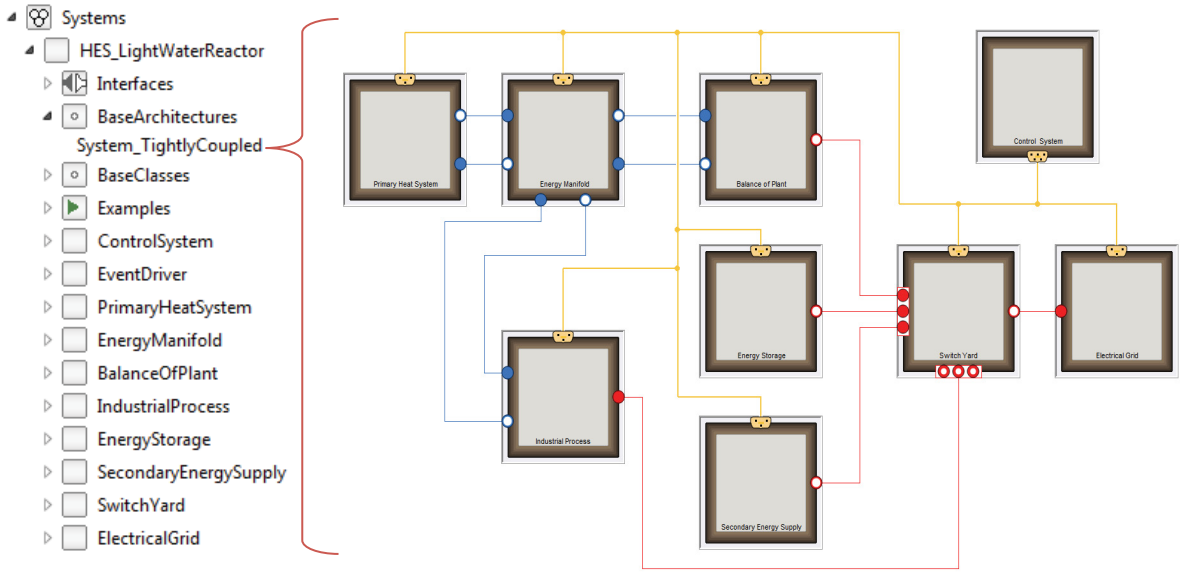


Figure 10. Tightly coupled NHES layout clearly defining the boundaries of each subsystem.

Each subsystem is developed within its own properly labeled package (Figure 11). Each of the subsystem packages contains a "BaseClasses" package, which defines the high-level "icon" interface shown in Figure 10. Another important feature of the subsystem package is the creation

of a signal sub bus. Each subsystem has its own unique signal bus that is used for the "Partial" models of the subsystem model, control system, and event drive. Defining each subsystem with its own signal bus enables clear identification of which signals (e.g., control signals) belong to which subsystem with the automatic connection creation ability of Modelica. Each of the subsystem signal busses can then be integrated into a master signal bus for system level cases.

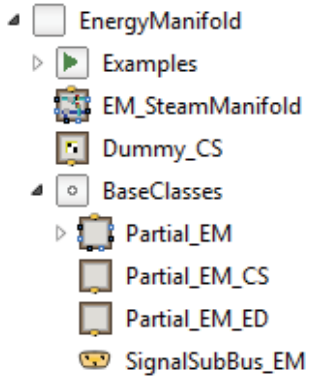


Figure 11. Representative organization of each subsystem, including the partial models and subsystem signal bus.

The baseline NHES under development is referred to as the "tightly coupled" case. As shown in Figure 10, this case consists of several subsystems, which integrate thermal energy and electricity to ultimately deliver an industrial product to a market and electricity to the grid. A brief description of each subsystem is taken from [15]. The overall system integration and modeling is performed by ORNL with subsystem model contributions from ANL, INL, and ORNL.

Primary Heat System

The primary heat system (PHS), or nuclear steam supply system, is based on the International Reactor Innovative and Secure (IRIS) integral pressurized light water medium power (1000 MWt) reactor design. The IRIS design was developed by an international consortium led by Westinghouse Electric Company. Originally sponsored by the Department of Energy as part of the Nuclear Energy Research Initiative program, IRIS was subsequently selected as an International Near Term Deployment reactor within the Generation IV International Forum (GIF) activities [15]. Given the relative maturity of the design and amount of quality information that is publically available, the IRIS concept was deemed an appropriate choice for development of a dynamic system model. The PHS model is provided by ORNL.

Energy Manifold

The energy manifold is the interface subsystem between the PHS and other subsystems. The name "manifold" is used because the role of the component is to distribute heat from the PHS among multiple subsystems. The initial thermal manifold that will be considered is a steam manifold. This manifold consists of a valve system that switches flow between subsystems based on demand signals. The energy manifold model is provided by ORNL.

Balance of Plant

The balance of plant uses the flow from the energy manifold and generates the primary source of electricity in the NHES model. The flow is then sent back to the energy manifold after being properly conditioned (e.g., heated/cooled, pumped). The balance of plant model is provided by ORNL.

Energy Storage

The initial energy storage subsystem consists of banks of batteries corresponding to commercially available types. The storage subsystem size is scaled according to the power storage demands of the system under investigation by varying the number of batteries connected in parallel. Important electrochemical behaviors, such as discharge rates and lifetime limitations, will be included in the battery models. The battery model will also be able to charge and discharge according to the demands of the supervisory control system. The energy storage battery model is provided by ANL.

Industrial Process

A steam electrolysis hydrogen production plant is the initial industrial process that will be included in the hybrid energy system optimization study. Steam from the PHS (via the energy manifold) provides a portion of the process heat required. The steam will then be sub cooled and sent back to the energy manifold. The remainder of the power demands will be fulfilled by an electricity supply (i.e., steam generator, battery, gas turbine, grid). The hydrogen plant industrial process model is provided by INL.

Secondary Energy System

A natural gas turbine model will be included in the overall hybrid energy system as the secondary energy supply. The gas turbine will cover rapid grid demands that the dynamics of the remainder of the hybrid system cannot handle. The gas turbine secondary energy storage model will be provided by INL.

Switchyard

All electrical connections from various subsystems (e.g., balance of plant, electrical storage system, etc.) will be routed through the switchyard. The switchyard will contain the necessary logic to appropriately connect the hybrid system to the grid as well as balance electricity demands within the hybrid system (e.g., battery and industrial process demands). The switchyard model is provided by ORNL.

Control System

Each subsystem will have an accompanying control system. These control systems will then be placed within the overall control system shown in Figure 10.

2.4.2 Resources

The Resources folder contains files related to the models in NHES that are not managed directly by the Dymola modeling environment. This currently includes (but is not limited to):

- *Images* used to decorate model block diagrams
- *Scripts* that define unit tests for various models within the NHES model tree. The automated test system is able to build, run, and collect the results from models based on the contents of these files.
- *ReferenceResults* contains output for test scripts that have been determined by model developers to be correct. The outputs from future runs of test scripts are compared against those stored in this folder; any differences cause the test to fail.

It is the experience of the team that when such files are stored in a Modelica package it is possible for them to be deleted.

2.4.3 Testing

This folder contains software used to perform automated unit testing of contributed models in the NHES folder. The project has adopted *BuildingsPy* [16], which is an open source project developed in the Python scripting language by Lawrence Berkeley Laboratory (LBL) to:

- Run Modelica simulations using Dymola in an automated manner
- Read and process output from such simulations
- Generate and run unit tests from scripts written by developers

BuildingsPy was developed as part of the Department of Energy (DOE) *Modelica Buildings Library* project [17], which applies the Modelica modeling language to Heating, Ventilation, and Air Conditioning (HVAC) systems and building controls. The HYBRID Modeling and Simulation project will take advantage of this existing automated test capability to assure the performance of its own models.

A copy of *BuildingsPy* will become part of the HYBRID repository in the Testing folder. This way any modifications of the testing library necessary to support project needs may be controlled and made available to all project contributors. The function of the test system is discussed in section 5.

2.4.4 RAVEN Submodule

As indicated above, the RAVEN code will play an important part in analyzing the economics of hybrid energy systems for this project. For the convenience of project participants, RAVEN has been added to the HYBRID repository structure as a Git submodule. If a HYBRID project developer needs to run RAVEN for their work, they may use a Git submodule command to automatically download it and its dependencies into the "raven" folder in the hybrid directory:

```
cd hybrid
```

```
git submodule update –init --recursive
```

RAVEN will not be loaded unless the developer initiates it using the above command. The folder name “raven” is fixed because RAVEN is under active development and is stored in its own Git repository with that name. When included as a submodule, Git may be used to keep it up to date. Access to RAVEN is controlled; hence, any project developer needing it must request it.

3. SharePoint, Mailing List, and Developer Meetings

The HYBRID modeling and simulation project involves teams from different national laboratories. In order to facilitate communication, several measures have been implemented.

First, a biweekly videoconference meeting using the “BlueJeans” technology has been set up. The BlueJeans technology allows the different participants to see each other over video. It supports a large number of operating systems including videoconference room equipment that already exists at each laboratory, but it also supports personal computers, tablets, and smartphones. Furthermore, BlueJeans allows the team to share presentations and videos remotely [18]. During the videoconference meeting, developers can report their progress and issues. As required by the agile development method (described in section 4.1), the project priorities are evaluated and new ideas, bugs, etc. are discussed. In addition to the biweekly videoconference meetings, periodic face-to-face meetings are organized. In these face-to-face meetings, all developers and management meet at one of the three national laboratories involved in the project. These are longer, usually two day meetings, where each development team member gives a presentation on his or her progress and high level strategy decisions are discussed with management. The first of these technical meetings was sucessfully been held at INL, May 9-11, 2016.

A second means of communication is the developer email list. This list allows any member of the team to send email to the collective of the project developers and managers. The email list can be used for various things; for example, to quickly generate a discussion with a broad group of people regarding a (technical) problem a developer encounters. The developer email list is also a communication channel to broadly distribute project relevant information. The list email address is:

hybrid-devel@inl.gov

Third, to exchange documents quickly, a SharePoint directory has been established [19]. SharePoint allows teams and organizations to intelligently share and collaborate on content from anywhere and on any device. The HYBRID SharePoint is used to distribute presentation material, but also to collaborate on reports and other documents. The HYBRID modeling and simulation SharePoint is located at:

https://art.inl.gov/hybridenergy/HES_Private/test/

4. Workflow

There are two main software development strategies and associated workflows: waterfall and agile. The agile strategy is most appropriate to the current application because the HYBRID workflow is driven by the requirement for new capabilities. This section describes the agile software development strategy, discusses the quality assurance requirements for HYBRID, and presents the adopted workflow and minimum documentation requirements.

4.1 Agile versus Waterfall

As mentioned, there are two main software development strategies: waterfall and agile. Much like manufacturing workflows, the waterfall methodology is a sequential design process. This means that as each of the workflow stages are completed (conception, initiation, analysis, design, construction, testing, implementation, and maintenance), the developers then move on to the next step. The advantages of this approach are that size, cost and timeline of the project are clear from the beginning. However, because this process is sequential, once a step has been completed developers cannot go back to a previous step – not without scratching the whole project and starting from the beginning. There is no room for change or error, so a project outcome must be clearly defined and an extensive plan must be set in the beginning and then followed carefully [20].

Agile came about as a "solution" to the disadvantages of the waterfall methodology. Instead of a sequential design process, the agile methodology follows an incremental approach. Developers begin with a simplistic project design, and then begin to work on small modules. The work on these modules is done in "sprints." At the end of each sprint, project priorities are evaluated and tests are run. These sprints allow for bugs to be discovered, and customer feedback to be incorporated into the design before the next sprint is run. An issue tracking system and regular meetings among all developers are required for a successful implementation of agile.

The HYBRID project naturally splits into "modules" or components that can be developed independently. Furthermore, the scope of the HYBRID project may grow in the future. The agile approach is therefore more suitable to accommodate the needs of the HYBRID modeling and simulation development project.

4.2 Quality Assurance

To understand what level of software quality assurance (QA) DOE requires for the HYBRID projects, one must first ask if the developed software fulfills any safety functions [21]. According to DOE definition, "safety software" is software that includes the following:

- *Safety system software*: Software for a nuclear facility that performs a safety function as part of a structure, system, or component and is cited in either (a) a DOE-approved documented safety analysis, or (b) an approved hazard analysis per DOE P 450.4, "Safety Management System Policy," dated October 15, 1996 (or latest version), and 48 CFR 970-5223.1, "Integration of Environment, Safety, and Health into Work Planning and Execution."

- *Safety analysis and design software*: Software that is used to classify, design, or analyze nuclear facilities. This software is not part of a structure, system, or component, but helps to ensure that proper accident or hazards analysis of nuclear facilities or a structure, system, or component that performs a safety function is conducted.
- *Safety management and administrative controls software*: Software that performs a hazard control function in support of nuclear facility or radiological safety management programs, technical safety requirements, or other software that performs a control function necessary to provide adequate protection from nuclear facility or radiological hazards. This software supports eliminating, limiting, or mitigating nuclear hazards to workers, the public, or the environment as addressed in 10 CFR 830 ("Nuclear Safety Management"), 10 CFR Parts 830, "Nuclear Safety Management," the Department of Energy Acquisition Regulation (DEAR) Integrated Safety Management System (ISMS) clause, and 48 CFR 970-5223.1.

It is assumed that, at the present stage, the HYBRID modeling and simulation framework can be classified as non-safety software. Furthermore, similar to MOOSE applications that are classified as "research and development," it is assumed that also the HYBRID software framework falls under this category. The associated quality level is Quality Level 3 (QL3) software. This designation imposes low requirements on quality control, testing, and documentation during the early development period. The quality level could change as the application development continues.

4.3 HYBRID Workflow

As discussed previously, due to the modular nature and evolving scope of the HYBRID modeling and simulation project, the agile software development strategy is applied. The QL3 designation and the fact that HYBRID is assumed to be non-safety software provides a lot of freedom in defining a workflow from a QA standpoint. Figure 12 shows the agile defined workflow for the HYBRID project. The workflow is an adjustment from the widely used agile "integrated workflow" [22].

4.3.1 New Issue and Change Control Board

The first step in the workflow is that a user or developer submits a tracking issue, i.e. a ticket. The tracking issue can be anything from a bug to a new capability needed, a model improvement or an entirely new model. The next step is that the "change control board" (CCB) reviews the submitted issue. The CCB will decide if the suggested issue is needed by the project and if it will or will not be implemented. For the HYBRID project, the CCB is the collective of all developers. The biweekly developer meeting includes the discussion and review of newly submitted issue tickets.

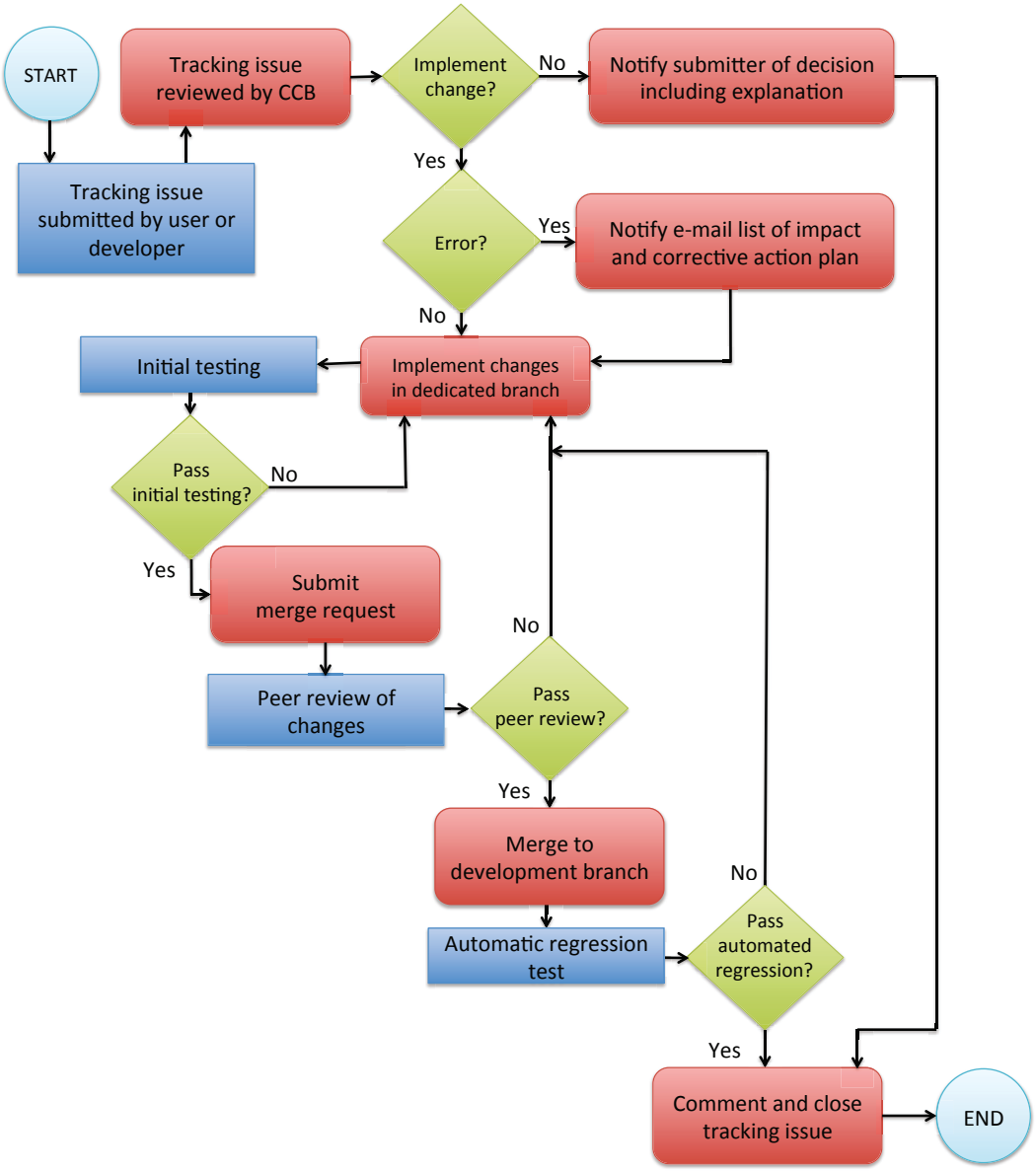


Figure 12. HYBRID modeling and simulation software development workflow.

If the CCB decides to reject a ticket, the submitter is notified of the decision and an explanation as to why the ticket has been rejected is provided. This explanation is also added to the tracking system, i.e. inside the ticket. Additional comments are also recorded in the ticket and the ticket is then closed.

If the CCB decides that the ticket will be implemented, a priority and a developer are assigned. If a developer proposes a new feature or model, normally this developer is assigned to the ticket, but if the ticket is a bug report from a user, a developer has to be assigned. If the issue is an error, i.e. a bug that has been discovered in existing code, the users and developers are informed about the impact of the error and the corrective action plan via the developer email list. The ticket implementation then begins.

4.3.2 Implementing Changes: Central and Branch Repositories

The project has a central repository that is considered as the official repository, which is managed by the project maintainers. Developers clone this repository to create identical local copies of the code base. Source code changes in the central repository are periodically synchronized with the local repository. The developer creates a new branch in his local repository and modifies source code on that branch. Periodically, the changes need to be integrated into the central repository. This is called "push" changes back to the central repository. This creates a copy of the local branch in the central repository for other developers to check out. Note, however, that this should not be confused with a "merge request," where code from a branch is merged into another branch. Frequently, in a distributed team, each developer has write access to his or her own public repository and has read access to everyone else's repository [23].

The distributed model is generally better suited for large projects with partly independent developers, because developers can work independently and submit their changes for merge (or rejection). The distributed model flexibly allows adopting custom source code contribution workflows.

In the HYBRID project, the "devel" branch contains all reviewed developments. To start development on a ticket, the developer first clones the repository onto his local machine and then creates a development branch from the "devel" branch. If the developer is going to help work on an issue that somebody else is already working, he does not need to create new branch, but can check out the existing branch associated with the issue. This requires that the branch has previously been "pushed" to the central repository by the original developer who created it. On the command line, the following Git commands can be used to achieve the discussed operations. These instructions can also be found in the HYBRID wiki.

- ***Clone or update the repository***: The developer first needs to clone the repository on his local machine:

  ```
  git clone git@hpcgitlab.inl.gov:hybrid/hybrid.git
  ```

  Alternately, if he or she already has a clone, the clone just needs to be updated:

  ```
  cd hybrid
  git pull
  ```

- ***Create a new development branch***: First, a ticket needs to be created in GitLab under "Issues" describing the development. For this, click on "Issues" on the left and then "+ New Issue" on the top right. After the ticket has ben approved and assigned, a new branch of the repository from "devel" can be created:

  ```
  cd hybrid
  git checkout devel
  git checkout -b username/branch_name
  ```

- ***Download an existing development branch***:

  ```
  cd hybrid
  git fetch
  git checkout username/branch_name
  ```

- ***Do development***: To edit files use:

  ```
  git add -u
  ```

  To create new files in the repository, use

  ```
  git add new_file git commit
  ```

- ***Push***: Note that for the subsequent push commands to work, the configuration

  ```
  git config --global push.default simple
  ```

  has to be used.

  For the first push, use

  ```
  git push --set-upstream origin username/branch_name
  ```

  After the first push, then:

  ```
  git push
  ```

  can be used.

The developer is required to provide test cases together with each development. These tests should access (and test) as much of the developed code as possible. Once the development is finished, the developer must then run all the test cases that already exist from other developments and has to make sure that none of the expected results for these tests have changed. This is a way to check that the code is non-regressive and the new development did not break any existing functionality. These tests are usually automated. Once the new development passes all of the tests, the developer can issue a "merge request."

4.3.3 Merge Requests

Contributions to a source code repository that uses a distributed version control system are made by means of a "merge request." The developer requests that his development be merged into the "devel" branch. The developer creates the request to notify the repository maintainer that his development is finished. A comment thread is associated with each merge request. This allows for focused discussion of code changes. Submitted merge requests are visible to anyone with repository access. A reviewer is then assigned to the merge request. The reviewer checks that the development is made according to the code standards (see section 6) and sufficient tests

are submitted. If the reviewer's background allows it, he can also review the code from a technical perspective. It is preferred to have a technical review, but it's not always possible to find a reviewer within the project with the same technical background than the developer, since they are highly specialized in their respective fields. The reviewer decides if the merge request should be accepted or rejected.

Once the merge request is reviewed and approved, it is merged into the "devel" branch. The code is then tested with an automatic test suite (see regression test, section 5). The reviewer checks that any new code has appropriate test coverage. If the automatic regression test suite that runs in the "devel" branch completes successfully, the reviewer documents and comments the ticket and then closes it.

4.4 Minimum Documentation Required

Every new development needs to be documented. It is required that the code is documented "in line," i.e. with comments according to the defined code standards (see section 6). In addition, every new development needs a paper documentation that describes the models used as well as a "user guide" that explains how others can use the development.

5. Regression Testing Framework

Regression testing is a development methodology that is used to verify the correct performance of software after it is changed. As software is created, developers create additional small programs (*tests*) to exercise its various features. The tests and their expected results are stored as part of the repository. Before new software changes are accepted into the repository, all of the tests are run against the modified code. The outputs of these tests are then compared against expected results stored in the repository. If a result is different than expected for any test, or if any test does not run successfully, then the test is considered to have failed. When the tests do not pass, a developer must determine why the failure occurred before a proposed change may be accepted. Regressions tests are one way to assure the quality of software produced.

The software product from the HYBRID project will include models developed in the Modelica language under the Dymola simulation environment. Regression testing may also be applied to such models. After initially considering developing a specific testing framework for Modelica models, the project developers identified and adopted the *BuildingsPy* [16] open source software package from LBL to perform this testing. Written in the Python scripting language, *BuildingsPy* was developed by LBL to provide automated testing for the Modelica models developed for their own *Modelica Buildings Library* [17]. The library contains Python code that:

- Automates the process of loading, building, and executing a Modelica model in the Dymola-run Modelica simulations using the Dymola modeling environment. *BuildingsPy* also determines if such processes were successful or not.
- Collects, compares, and plots results from Modelica simulations, if desired. Python libraries *matplotlib* and *scipy* are utilized to provide these functions.

- Provides a way for developers to create one or more tests for a Modelica package. A test is defined by identifying a model to run and selecting one or more variables from the model results to compare against reference results.
- Contains functions that assist in the restructuring of Modelica code packages (these features are not currently used by the HYBRID project).

After *BuildingsPy* was adopted for regression testing of Modelica models, project contributors at ORNL provided enhancements that increase its usefulness to the project. These additional features are:

- The ability to specify the location of the folder that contains the test specifications at an arbitrary place in the file system (The original *BuildingsPy* requires this to be located in the top folder of the Modelica model package under test).
- A way to have the test framework run a single test selected by name instead of the default behavior of running all tests found in the Resources directory.

This ORNL-modified version of *BuildingsPy* is named *ModelicaPy*, and will become part of the HYBRID repository in the Testing folder. This way any further modifications of the testing library needed to support the project may be controlled and made available to all project contributors.

5.1 Testing System Prerequisites

The test system has been verified to function on either Windows™ or Linux systems, providing the following conditions are met:

- The Dymola simulation environment must be installed on the system, and the command "dymola" must be in the current system command search path.
- A fully licensed version of Dymola must be used—the demonstration version cannot be used to build complex models.
- The Python scripting language environment must also be installed on the system, and Python packages matplotlib.pyplot and scipy.io must be present. These packages are used to display plots and to read data from Dymola simulation output files. For Windows™, installing the Anaconda Python distribution fulfills all three of these requirements.

5.2 Test Definition

Model developers define tests by creating Modelica script files (those having the *.mos* extension) that reside in the *Resources/Scripts/Dymola* folder (and subfolders). These scripts contain commands that ModelicaPy will scan for when the tests are run. An example script file is presented in Figure 13 to illustrate the format of a test specification.

```
simulateModel("NHES.Nuclear.ReactorKinetics.Examples.PointK
inetics_Test", stopTime=10, numberOfIntervals=1000,
method="dassl", resultFile="PointKinetics_Test");

createPlot(id=1, y={"reactorKinetics.Q_total"}, grid=true);
```

Figure 13. Sample test script file.

A Modelica model package may have many tests associated with it. Each of these tests is contained in its own Modelica script file. Each test specification file must be named with the extension *.mos* to enable it to be found by ModelicaPy when it runs. These files describe which model within the package to simulate and what output variables from the executed model should be compared against stored results.

Each test file must contain one *simulateModel* command that specifies what model in the package to run and in what manner. In the above example test, the output of the model called *NHES.Nuclear.ReactorKinetics.Examples.PointKinetics_Test* will be examined. Simulation parameters may also be specified in the test script *simulateModel* command. Such parameters include simulation start and stop time, and computation method, interval, and tolerance. Refer to the *Dymola User Manual* [4] for a complete description of *simulateModel* command parameters.

The test script also contains one or more *createPlot* commands that specify which variables will be read and compared against stored values to assess whether the output is the same or not. A mismatch between computed and stored values may indicate that a change elsewhere in the model package has caused an unintended regression problem. The above example shows that one variable *reactorKinetics.Q_total* will be collected and compared to determine if the output has remained consistent with the stored value.

A test is considered a failure under any of the following conditions:

- The model specified in the *simulateModel* command fails to build and run.
- The values of any model variables specified for comparison do not match the values previously stored values designated for comparison.
- There are no values stored for a model variable to compare against.

5.3 Running the Tests

The test system is accessed by running a Python language script located in the top directory of the HYBRID project repository. This file *run_tests.py* provides several parameters to control its function. The format of the command is:

```
python run_test [-h | -help] [-i | -interactive]
                [-g | -gui ] [-s | -st <single test name>]
                [-p | -proc <number of processes>]
```

Where:

-h or –help	Causes the script to print the help text shown in Figure 14 and exit without running any tests.
-i or –interactive	Runs the tests in interactive mode, which is used primarily to set the expected value for tests. When run in this mode, any deviations in results from the stored expected values will be displayed in a plot. In such a case (or if there are no expected results found) the user will be asked if they should be stored as the new expected result.
-g or –gui	Causes the Dymola graphical user interface to be shown when building and simulating the models under test (the default is for the GUI to be hidden). This may be useful when the test script is failing for an unknown reason.
-s or –st <single test name>	Tells the test script to run a single test instead of executing all tests found. Specifies the name of the desired test without the .mos extension.
-p or –proc <number of processes>	When specified, limits the number of concurrent test processes to the number provided. The default is to use all available processors on the computer on which it is being run. This is useful to limit Dymola license consumption.

```
#-----------------------------------------------------------------------------------------
#
#  Hybrid Systems Test Script (run_tests)
#
#  Makes extensive use of 'BuildingsPy', part of the Modelica Buildings Library
#  (http://simulationresearch.lbl.gov/modelica/buildingspy/) as modified by Scott
#  Greenwood of Oak Ridge National Laboratory (ORNL).
#
#  Command Format:
#
#    run_test [-h | --help] [-g | --gui] [-i | --interactive]
#             [ -s | --st <single test name>] [-p | --proc <# procs>]
#
#  Where:
#    '-h' or '--help' shows this help text
#    '-i' or '--interactive' runs buildingpy tests in interactive mode.  In this mode deviations
#                            from expected test outputs will be shown using plots and the user
#                            is prompted to optionally set the test output as the new expected
#                            values.
#    '-g' or '--gui' will display the Dymola graphical user interface while test runs are in
#                    progress.
#    '-s' or '--st' <test name> causes only that test whose script file name is given to be run.
#                               (case sensitive, '.mos' is not necessary)
#    '-p' or '--proc' <number of processes> limits maximum number of processes (default is
#                                           to use all available).
#
#  Prerequisites:
#
#    - The command 'dymola' must be in your PATH
#    - Python, and Python packages 'matplotlib.pyplot' and 'scipy.io' need to be available
#        (Required by BuildingsPy)
#    - The Modelica package containing the models is contained in the NHES directory below
#        the one containing this script
#    - The .mos files specifying what to test are contained in the Resources/Scripts/Dymola
#        directory below NHES
#
#-----------------------------------------------------------------------------------------
```

Figure 14. Test script help message.

As *run_tests.py* executes, it generates text output such as that shown in Figure 15. Messages indicate when the test was run and with what parameters, as well as information about the models and its pass/fail status.

5.4 Repository Test System (Civet)

The primary purpose of having a single top-level test script is to provide an automated means to confirm that modifications to the model repository preserve expected behavior. INL has developed a tool called *Civet* to provide an automated connection between Git repositories and regression test systems. The HYBRID project is just one of many software development projects at INL making use of this capability.

```
################################################################
#
#  HYBRID SYSTEMS MODELICA TESTING
#  Run: 2016-09-14 15:02:38.361151
#
#  Current Directory:  /home/bobk/projects/hybrid
#  Using at most 2 processor(s)
#
#  Searching for ModelicaPy in: /home/bobk/projects/hybrid/Testing/ModelicaPy
#
Directory with .mo /home/bobk/projects/hybrid/NHES/package.mo
Directory with Resources is /home/bobk/projects/hybrid/NHES
Beginning to run tests
Executable is on path
Current working directory is good
Using  2  of  24  processors to run unit tests.
Number of models   :  140
        blocks   :  1
        functions:  52
Preparing simulations
Temp Directories set... preparing to write runscripts...
Generated  5  regression tests.

check reference points...
*** Warning: Output file of PHS_ED_Test.mos is excluded from result test.
Deleting Temp Directories..
Checking for errors...

Number of models with unspecified initial conditions        : 1

Script that runs unit tests had 4 warnings and 0 errors.

See 'simulator.log' for details.
Printing list of files that may be excluded...
Execution time = 159.185 s
#
#  BuildingsPy Test Framework Returns:  2  (PASSED WITH WARNINGS)
#
################################################################
```

Figure 15. Sample test output.

As previously discussed in Section 2, the Modelica models for the HYBRID project are stored in a Git repository at INL. When a developer completes a unit of work, such as a new component model or modification to an existing model, a *merge request* is created by the developer to indicate that some changes to the repository are ready to be considered for inclusion in the official repository branch *devel*. When Civet is configured for a repository, a merge request will trigger automated testing of the repository including the proposed changes.

That testing proceeds as follows:

1. The HYBRID test job triggered by the merge request is placed in a queue on the Civet server.
2. When a Civet client having the resources to run HYBRID model tests becomes available, it claims the job. (These prerequisites are discussed in Section 5.1)

3. The client receives the identifier for the repository version containing the proposed changes and uses it to retrieve the branch. This branch contains the updated models, test system, test scripts, and expected results.
4. The client then runs the top-level test script *run_tests.py* described in Section 5.3. This script will return a single value to the Civet client indicating whether there was a failure or not. The text output of the script is also collected and returned to the Civet server where it can be monitored by developers.
5. The success or failure of the automated tests is displayed on the web interface used by developers to process merge requests. This interface is also configured to prevent merge requests with failing tests from being accepted.
6. If the tests fail, the developer that submitted the merge request is responsible for determining why it failed and submitting an updated branch that corrects the issue.
7. When a branch corresponding to a merge request is updated by the developer, this procedure begins again with tests repeated on the new files.

Regression testing provides a consistent method to ensure that incremental software changes to a large code base causing unintended consequences are quickly found and corrected. The project has chosen to use the Modelica testing features found in the modified *BuildingsPy*, called ModelicaPy, to provide this capability. INL's Civet tool provides the means to run these tests in an automated manner and integrate their results into the HYBRID code repository and issue tracking system.

6. Modelica Code Standards

This section provides a checklist that should be used when contributing a new class (model, block, connector, function, package, etc.) to the libraries that comprise NHES models.

6.1 General

1. Follow the conventions of the Modelica Standard Library (MSL), which are as follows:
   Note, in the html documentation of any Modelica library, the headings "h1, h2, h3" should not be used. These headings are reserved for the automatically generated documentation and headings. Additional headings in the html documentation should start with "h4."

   In the Modelica package, the following conventions are used:

   a. **Comments and annotations** always start with a capital letter, e.g.,
   ```
   parameter Real a = 1 "Arbitrary factor";
   ```

   b. **Class and instance names** are usually written in upper and lower case letters, e.g., "ElectricCurrent". An underscore is only used at the end of a name to characterize a lower or upper index, e.g., "pin_a" may be rendered as "pin_a".

   c. **Class names** always start with an upper case letter.

   d. **Instance names**, i.e., names of component instances and variables (with the exception of constants), usually start with a lower case letter with only a few exceptions based on common terminology (such as "T" for a temperature variable).

e. **Constant names**, i.e., names of variables declared with the "constant" prefix, follow the usual naming conventions (upper and lower case letters) and usually start with an upper case letter, e.g. UniformGravity, SteadyState.

f. The two **connectors** of a domain that have identical declarations and different icons are usually distinguished by "_a", "_b" or "_p", "_n", e.g., Flange_a/Flange_b, HeatPort_a, HeatPort_b.

g. The **instance name** of a component is always displayed in its icon (= text string "%name") in **blue color**. A connector class has the instance name definition in the diagram layer and not in the icon layer. **Parameter** values, e.g., resistance, mass, gear ratio, are displayed in the icon in **black color** in a smaller font size than the instance name.

h. A **connector class** has the instance name definition in the diagram layer and not in the icon layer.

i. A main package usually has the following subpackages:

- **UsersGuide,** containing an overall description of the library and how to use it.
- **Examples,** containing models demonstrating the usage of the library.
- **Interfaces,** containing connectors and partial models.
- **Types,** containing type, enumeration, and selected definitions.
- **BaseClasses,** containing models, partial models, etc. that are not of interest to the user.

In addition to the MSL conventions, the following conventions are used:

j. **Names of models, blocks and packages** should start with an upper-case letter and be a noun or a noun with a combination of adjectives and nouns. Use camel-case notation to combine multiple words, such as HeatTransfer.

k. **Parameter and variables** names are usually a character, such as T for temperature and p for pressure, or a combination of the first three characters of a word, such as higPreSetPoi for “high pressure set point”.

l. **Comments** should be added to each class (package, model, function, etc.). The first character should be an upper case letter.

m. Where applicable, all variables, including protected variables, must have units.

2. All classes, with the exception of models within BaseClasses and constants, must have icons.
3. Examples, i.e., regression tests, should be in a directory such as Electrolysis.Examples.
4. Do not copy sections of code. Use object inheritance.

6.2 Type Declarations

1. Declare all public parameters before protected ones.
2. Declare variables and final parameters that are not of interest to users as protected.
3. Set default parameter values as follows:
   a. If a parameter value can range over a large region, do not provide a default value. Examples are nominal mass flow rates.
   b. If a parameter value does not vary significantly but needs to be verified by the user, provide a default value by using its starting attribute. For example, for a heat exchanger, use
   ```
   parameter Real eps(start=0.8, min=0, max=1, unit="1")
   "Heat exchanger effectiveness";
   ```
   Do not use
   ```
   parameter Real eps=0.8(unit="1")
   "Heat exchanger effectiveness";
   ```
   as this can lead to errors that are difficult to detect if a modeler forgets to overwrite the default value of 0.8 with the actual value. The model will simulate, but gives wrong results due to unsuited parameter values and there will be no warning. On the other hand, using `parameter Real eps(start=0.8)` will give a warning and, hence, users can assign better values.
   c. If a parameter value can be precomputed based on other parameters, set its value to the appropriate equation. For example,
   ```
   Parameter Medium.MassFlowRate m_flow_small(min=0) =
   1E-4*m_flow_nominal;
   ```
   d. If a parameter value should not be changed by a user, use the `final` keyword. For example, use
   ```
   final parameter Modelica.SIunits.Frequency fn=60
   "Nominal frequency";
   ```
4. For parameters and variables, provide values for the min and max attributes where applicable. Be aware that these bounds are not enforced by the simulator. If the min and max attributes are set, each violation of these bounds during the simulation may raise a warning.

   Compilers may allow suppression of these warnings. In Dymola, violation of bounds can be checked using
   ```
   Advanced.AssertAllInsideMinMax=true;
   ```
5. For any variable or parameter that may need to be solved numerically, provide a value for the `start` and `nominal` attribute.
6. Use types from `Modelica.SIunits` where possible.

6.3 Equations and Algorithms

1. Avoid events (i.e., discrete behaviors that are generated by conditional expressions) where possible.
2. If possible, only divide by quantities that cannot be zero. For example, if x may equal zero, use y=x, not 1=y/x, as the latter version indicates to a simulator that it is safe to divide by x.
3. Use the assert function to check for invalid values of parameters or variables. For example, use assert(phi>=0, "Relative humidity must not be negative.").
4. For computational efficiency, equations shall, were possible, be differentiable and have a continuous first derivative.
5. Avoid equations where the first derivative with respect to another variable is zero. For example, if x, y are variables, and x = f(y), avoid y = 0 for x<0 and y=x^2 otherwise. The reason is that if a simulator tries to solve 0=f(x), then any value of x <= 0 is a solution, which can cause instability in the solver. Note that this problem does not exist for constant functions, as their first derivate will replaced due to optimization within the solver.
6. Do not replace an equation with a constant that has a single value unless the derivative of the original equation is zero for this value. For example, if computing a pressure drop dp may involve computing a long equation, but one knows that the result is always zero if the volume flow rate V_flow is zero, one may be inclined to use a construct of the form dp = smooth(1, if V_flow == 0 then 0 else f(V_flow));. The problem with this formulation is that for V_flow=0, the derivative is dp/dV_flow = 0. However, the limit dp/dV_flow, as |V_flow| tends to zero, may be non-zero. Hence, the first derivative has a discontinuity at V_flow=0, which can cause a solver to fail to solve the equation because the smooth statement declared that the first derivative exists and is continuous.
7. Make sure that the derivatives of equations are bounded on compact sets. For example, instead of using y=sign(x) * sqrt(abs(x)), approximate the equation with a differentiable function that has a finite derivative near zero.

6.4 Package Order

1. Packages are first sorted alphabetically:
   Actuators
   Boilers
   Chillers
   HeatExchangers
2. After alphabetical sorting, the following packages, if they exist, are moved to the front:
   UsersGuide
   Examples
   and the following packages, if they exist, are moved to the end:
   Sources
   Sensors

Media
Interfaces
Types
Data
Utilities (functions, records, etc.)
Icons
BaseClasses

6.5 Documentation

1. Add a description string to all parameters and variables, including protected ones.
2. Group similar variables using the group and tab annotation. For example, use

```
parameter Modelica.SIunits.Time tau = 60
"Time constant at nominal flow"
annotation (Dialog(group="Nominal condition"));
```

or use

```
parameter Types.Dynamics substanceDynamics=energyDynamics
"Formulation of substance balance"
annotation(Evaluate=true, Dialog(tab = "Assumptions",
group="Dynamics"));
```

3. Add model documentation to the info section. To document equations, use the format

```
<p>
The polynomial has the form
</p>
<p align=\"center\" style=\"font-style:italic;\">
y = a<sub>1</sub> + a<sub>2</sub> x + a<sub>3</sub> x<sup>2</sup> + ...,
</p>
<p>
where <i>a<sub>1</sub></i> is ...
To denote time derivatives, such as mass flow rate, use
<code>m&amp;#775;</code>.
To refer to parameters of the model, use the format
To linearize the equation, set <code>linearize=true</code>.
To format tables, use
<p>
<table summary=\"summary\" border=\"1\" cellspacing=\"0\" cellpadding=\"2\"
style=\"border-collapse:collapse;\">
<tr><th>Header 1</th>  <th>Header 2</th>  </tr>
<tr><td>Data 1</td>  <td>Data 2</td>  </tr>
</table>
</p>
To include figures, place the figure into a directory in Electrolysis/Resources/Images/ that
has the same name as the full package. For example, use
</p>
<p align=\"center\">
```

```
<img alt=\"Image\"
src=\"modelica://Electrolysis/Resources/Images/Electrolyzers/Electrolyzer.png\"/>
</p>
<p>
```
To create new figures, put the source file for the figure, preferably in svg format, in the same directory as the png file. svg files can be created with http://inkscape.org/, which works on any operating system.

4. Add author information to the revision section.
5. Run a spell check.
6. Start headings with <h4>.
7. Add hyperlinks to other models using their full name. For example, use
   ```
   See <a href =\"modelica://Modelica.Fluid.Vessels.BaseClasses.VesselPortsData\">
   Fluid.Vessels.BaseClasses.VesselPortsData </a>.
   ```
8. To refer to names of parameters or variables in the documentation and revision sections, use the syntax < code >...</ code >. Do not use <tt>...</tt>.
9. Always use lower case html tags.

6.6 Functions

1. Use the smoothOrder annotation if a function is differentiable.

6.7 Regression Tests

1. Implement at least one regression test for each model and block, and run the regression tests. Regression tests should cover all branches of if-then constructs.

7. Conclusions

One of the goals of the HYBRID modeling and simulation project is to assess the economic viability of hybrid systems in a market that contains renewable energy sources such as wind and solar. The overarching concept is that it is possible for the nuclear plant in a nuclear-renewable hybrid energy system to sell products and commodities in addition to electricity. These alternative commodities absorb (at least partially) the volatility introduced on the grid by the variable renewable energy sources. Candidate systems are currently modeled in the Modelica programming language. To assess system economics, an optimization procedure is used to find the minimal cost for electricity production. The RAVEN code is used as a driver for the problem.

The modeling and simulation project involves teams from three national laboratories, namely ANL, ORNL and INL that develop the different Modelica component models as well as the RAVEN driver and optimization framework. To simplify and organize the development across the different laboratories, some computer infrastructure and development procedures have been established:

- The core of the software development infrastructure is the project management software **GitLab**. This software is a web-based Git repository manager with wiki and issue tracking features. The Git repository for the HYBRID modeling and simulation project is hosted on the INL HPC cluster. It is accessible under the following url: git@hpcgitlab.inl.gov:hybrid/hybrid.git. Access to the repository is controlled and every new collaborator to the project must request access to the repository.
- A **repository structure** has been established that defines where the developers have to store their models and test cases. The NHES directory hierarchy is derived from the existing standard format used for the Modelica standard library. The implementation of the actual nuclear hybrid energy system is performed within the "Systems" package that is organized to allow compartmentalized subsystem development, which enables clear boundaries of each system.
- It is assumed that, at this stage, the HYBRID modeling and simulation framework can be classified as non-safety "research and development" software. The associated quality level is **Quality Level 3** software. This imposes low requirements on quality control, testing, and documentation. The quality level could change as the application development continues.
- Despite the QL3 designation, a **workflow** for the HYBRID developers has been defined that include a coding standard and some documentation and testing requirements. The repository performs automated unit testing of contributed models. The project has adopted *BuildingsPy*, which runs Modelica simulation tests using Dymola in an automated manner and generates and runs unit tests from scripts written by developers.
- In order to assure **effective communication** between the different national laboratories a biweekly videoconference has been set-up fordevelopers to report progress and issues. In addition, periodic face-to-face meetings are organized to discuss high-level strategy decisions with management. A second means of communication is the developer email list. This is a list to which everybody can send emails that will be received by the collective of the developers and managers involved in the project. Third, in order to exchange documents quickly, a SharePoint directory has been set-up. SharePoint allows teams and organizations to intelligently share, and collaborate on content from anywhere.

8. References

1. C. Rabiti, et al., "Status on the Development of a Modeling and Simulation Framework for the Economic Assessment of Nuclear Hybrid Energy Systems", INL report INL/EXT-15-36451, September 2015.
2. C. Rabiti, et al., "Modeling, Simulation and Control Gap Analysis Report", INL report INL/EXT-15-34877, April 2015.
3. H. Elmqvist, et al., "Modelica—The Next Generation Modeling Language an International Design Effort", Proceedings of the 1st World Congress on System Simulation (WCSS'97), Singapore, September 1–3, 1997.

4. “CATIA Systems Engineering – Dymola”. Retrieved from http://www.3ds.com/products-services/catia/products/dymola
5. C. Rabiti, et al., “RAVEN, a new software for dynamic risk analysis”, Proceedings of PSAM 12, Honolulu, HI, 2014.
6. “Project management software”, (accessed September 3, 2016). Retrieved from https://en.wikipedia.org/wiki/Project_management_software
7. “Git”. Retrieved from https://git-scm.com/
8. “Version control”, (accessed September 24, 2016). Retrieved from https://en.wikipedia.org/wiki/Version_control
9. “Git”, (accessed September 23, 2016). Retrieved from https://en.wikipedia.org/wiki/Git_(software)
10. V. Driessen, (accessed January 05, 2010). “A successful Git branching model”. Retrieved from http://nvie.com/posts/a-successful-git-branching-model/
11. “Issue tracking system “, (July 17, 2016).Retrieved from https://en.wikipedia.org/wiki/Issue_tracking_system
12. C. Rabiti, et al., “RAVEN User Manual”, INL report INL/EXT-15-34123 Revision 5, February 2016
13. “Modelica Libraries “. Retrieved from https://www.modelica.org/libraries
14. “ThermoPower Modelica Library”. Retrieved from http://thermopower.sourceforge.net/
15. Oak Ridge National Laboratory, “Nuclear Hybrid Energy System FY16 Modeling Efforts at ORNL,” ORNL/TM-2016/418, August 2016.
16. “Modelica Buildings Library”. Retrieved from http://simulationresearch.lbl.gov/modelica/buildingspy/
17. “Modelica Buildings Library”. Retrieved from http://energy.gov/eere/buildings/downloads/modelica-buildings-library
18. “BlueJeans Features “. Retrieved from https://bluejeans.com/features
19. “SharePoint”. Retrieved from https://products.office.com/en-us/sharepoint/collaboration
20. “Agile & Waterfall Methodologies – A Side-By-Side Comparison”. Retrieved from http://www.base36.com/2012/12/agile-waterfall-methodologies-a-side-by-side-comparison/
21. INL, “Software Quality Assurance Plan for MOOSE and MOOSE-Based Applications”, INL report PLN-4005, March 16, 2016.
22. “Integrator workflow”, (accessed June 9, 2016). Retrieved from https://en.wikipedia.org/wiki/Integrator_workflow
23. “Distributed version control”, (accessed September 17, 2016). Retrieved from https://en.wikipedia.org/wiki/Distributed_version_control