**Jenkins Continuous Integration Example for the ISO 26262 Case Study Project**

User Guide

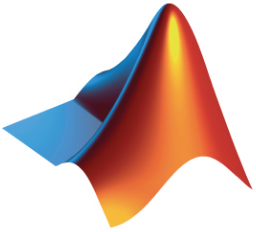


Table of Contents

[Introduction 3](#_Toc69754087)

[Description 3](#_Toc69754088)

[Operation Requirements 3](#_Toc69754089)

[Jenkins Installation and Service Setup 5](#_Toc69754090)

[Installing Jenkins 5](#_Toc69754091)

[Setting Up Jenkins Service with the Permission to Run MATLAB 5](#_Toc69754092)

[Overview of the Jenkins CI Framework 6](#_Toc69754093)

[Getting Started 7](#_Toc69754094)

[Creating a Jenkins Build Job for the ISO 26262 Project 7](#_Toc69754095)

[Testing the Build Job in MATLAB 8](#_Toc69754096)

[Launching the Build Job on Jenkins 8](#_Toc69754097)

[Task Definitions of the Jenkins Build Job 13](#_Toc69754098)

# Introduction

## Description

In addition to establishing a solid foundation for high-integrity software development using Model-Based Design methodologies, the ISO 26262 Case Study Project is equipped with an example framework to illustrate how to develop Jenkins build jobs, hence enabling all pertinent software verification activities to be fully automated via the practice of continuous integration. This Jenkins CI Framework is shipped with a ready-to-launch job definition that performs the following tasks:

* Generate a requirement report from each requirement set under the *ISO\_04\ISO\_4\_6\_5\_1\_TechSafReqSpec* and *ISO\_06\_06\_SwSafReq\WPs\ISO\_6\_6\_5\_1\_SwSafReqSpec* folders.
* Generate a system design description report from each model for software units/components under the *ISO\_06\_08\_SwU* folder.
* Verify each software unit model against high-level software requirements it implements.
* Check each software unit model for conformance to Software Model Standards.
* Analyze each software unit model for design error detection.
* Generate code from each software unit model.
* Check the code generated from each model for MISRA conformance.
* Verify the code generated from each model against high-level software requirements it implements.
* Generate low-level tests from each model if the coverage of high-level tests is incomplete.
* Verify the code generated from each model against low-level software requirements that are not covered by high-level tests.
* Assess the overall test coverage for verifying the code generated from each model.

## Operation Requirements

* Minimum MATLAB software requirements:
  + Automated Driving Toolbox™
  + Embedded Coder®
  + MATLAB Coder™
  + MATLAB Report Generator™
  + Model Predictive Control Toolbox™
  + Polyspace Bug Finder™
  + Simulink Check™
  + Simulink Coder™
  + Simulink Coverage™
  + Simulink Design Verifier™
  + Simulink Report Generator™
  + Requirements Toolbox™
  + Simulink Test™
  + Stateflow®
  + System Composer™
  + Vehicle Dynamics Blockset™
  + Control System Toolbox™
* External software requirements:
  + Supported C/C++ compiler per <https://www.mathworks.com/support/compilers.html>.
  + Jenkins™ with Summary Display Plugin and MATLAB Plugin

*Note:* [*Contact the MathWorks CI team*](https://www.mathworks.com/solutions/continuous-integration.html) *if you want to run CI on multiple hosts or in the cloud. Transformation products, such as MathWorks coder and compiler products, may require Client Access Licenses (CAL).*

# Jenkins Installation and Service Setup

## Installing Jenkins

Use the instructions below to install Jenkins:

1. Download Jenkins from <https://www.jenkins.io>.
2. Run the downloaded installer. Follow the setup wizard instructions to complete the installation.
3. Upon installation, Jenkins automatically launches and guides you through its configuration.
4. Once Jenkins is configured, you must sign in using the created credentials.
5. In Jenkins, open the Plugin Manager (click *Manage Jenkins > Manage Plugins*).
6. Locate both MATLAB Plugin and Summary Display Plugin under the *Available* tab of the Plugin Manager, and then install them.

## Setting Up Jenkins Service with the Permission to Run MATLAB

On Windows, the given instructions install Jenkins as a Windows service. By default, the Jenkins service launches using a local system account that does not have the permission to run MATLAB. You must restart the Jenkins service to use an account that has the proper privileges. Use the following procedure to start the Jenkins service with the necessary authorizations:

1. Open the Services App to locate the Jenkins service.
2. Right click *Jenkins*, and then select *Properties*. This opens the Jenkins Properties dialog.
3. In the Jenkins Properties dialog, click the *Log On* tab.
4. Under the *Log On* tab, select *This account*, and then enter your user account. Click *Browse* to select the user account if necessary.
5. Enter your account password, and then click *OK* to close the Jenkins Properties dialog.
6. Restart the computer.

# Overview of the Jenkins CI Framework

The Jenkins CI Framework is an extension of the ISO 26262 Project that streamlines the practice of continuous integration with the adoption of Model-Based Design (MBD). By leveraging MBD utilities in place, this framework provides a means to spawn off a sequence of software development and verification tasks that can be automated from a server.

The Jenkins CI Framework can be found under the *continuous\_integration* folder in the ISO 26262 Project. This folder has the following subfolders:

* In the *help* subfolder, you will find a user guide (this document).
* In the *job* subfolder, you will find the MATLAB sources of the Jenkins CI Framework.
* In *ISODemoTasks.m*, you will find a MATLAB class that outlines a Jenkins build job based on MBD operations in the ISO 26262 Project.
* In *JenkinsJob.m*, you will find an abstract class that serves as the basis for developing a Jenkins build job like *ISODemoTasks*.
* In *JenkinsReport.m*, you will find a MATLAB class that generates XML reports from a Jenkins build job for display purposes. These reports are formatted for use by the Jenkins Summary Display Plugin.
* In *runJob.m*, you will find a MATLAB function that serves as the entry point to launch a Jenkins build job.
* In jISODemoTasks.m, you will find the entry point to the Jenkins job
* In the *reports* subfolder, you will find the XML reports that are generated from running a Jenkins build job. This subfolder is initially empty.

# Getting Started

## Creating a Jenkins Build Job for the ISO 26262 Project

As a ready-to-use example, *ISODemoTasks* is a subclass of *JenkinsJob* that implements a complete Jenkins build job for running MBD tasks in the ISO 26262 Project. You can use it as a template for developing a Jenkins build job based on your actual project. Use the following tips to customize it.

Use the following guidelines when customizing *ISODemoTasks* to create a build job:

* Use the *TaskSequence* property to outline tasks that the job must run and the order in which they are carried out. Each entry of *TaskSequence* maps to a method that performs an MBD task. There are 14 such methods in *ISODemoTasks* (henceforth referred to as task methods). Each of these methods is named after the MBD utility under *tools\utilities* that it calls with a prefix of *task*. If you are not licensed to exercise all these 14 methods, edit *TaskSequence* accordingly to avoid running into errors. In the example below, the job performs only 3 tasks. It runs simulations to verify models against requirements, generates code from the models, and then runs SIL/PIL simulations to verify the generated code.

*TaskSequence = ["taskVerifyModel2Reqs"  
 "taskGenSrcCode"  
 "taskVerifyObjCode2Reqs"];*

* Use the *getModelNames* method to register the names of models you want the Jenkins build job to iterate on. Generically, this method discovers all model names from the model folders under *ISO\_06\_08\_SwU\WPs\ISO\_6\_8\_5\_1\_SwUnDesSpec.* Instead of performing a search, you can hardcode the model names in the method. Alternatively, you may use the *find\_mdlrefs* function to return model names from a known top-level model.
* Use the *getCompModelNames* method to register the names of models you want the Jenkins build job to iterate on for Component Integration tasks. Currently, only “taskCheckCompModelStds” is applied to these models to check conformance to modeling guidelines.
* Use the cell array called *allTopModels* in the *isTopModel* method to register the names of all known top-level models. All models are treated as referenced models unless otherwise registered here.

If necessary, you may create new task methods for the build job. Every task method must adhere to a set of general policies. When defining a new task method, keep in mind that it must:

* Exercise the appropriate MBD utility under *tools\utilities*.
* Use the *verifyOutcome* method to parse the return of the exercised utility in order to determine the execution outcome of each iteration, if applicable.
* Use the *verifyFile* method to determine if a report is successfully generated by the exercised utility for each iteration.
* Use an entry of the *TaskResults* property to collect results unpacked from all iterations of the task, if applicable.
* Use an entry of the *TaskOutcomes* property to capture the overall execution outcome of the task.
* Use an entry of the *TaskExceptions* property to capture any exception that may have been thrown when performing the task.

While not strictly enforced, it is recommended that you name each new task method after the utility it exercises for consistency reason. Do not forget to schedule the new tasks with the *TaskSequence* property.

## Testing the Build Job in MATLAB

For testing purposes, after creating a new build job as explained in the preceding section, you can run the job in MATLAB via the *jISODemoTasks* function. For example, to run the entire build job as outlined in *ISODemoTasks*, use the following command:

*>> jISODemoTasks*

You can also selectively test a specific task of the build job with a second argument when calling *runJob*. For instance, to execute only the *taskVerifyModel2Reqs* method of *ISODemoTasks*, use the following command:

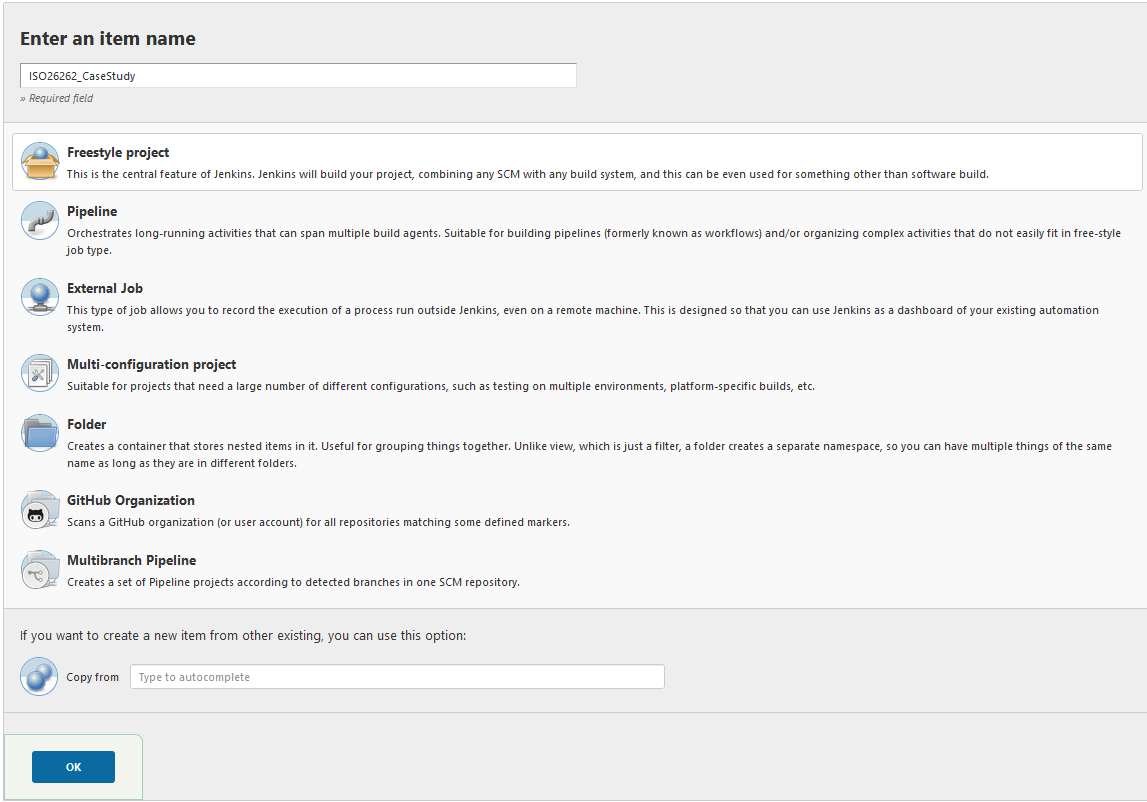
*>> runJob(ISODemoTasks, 'taskVerifyModel2Reqs')*

## Launching the Build Job on Jenkins

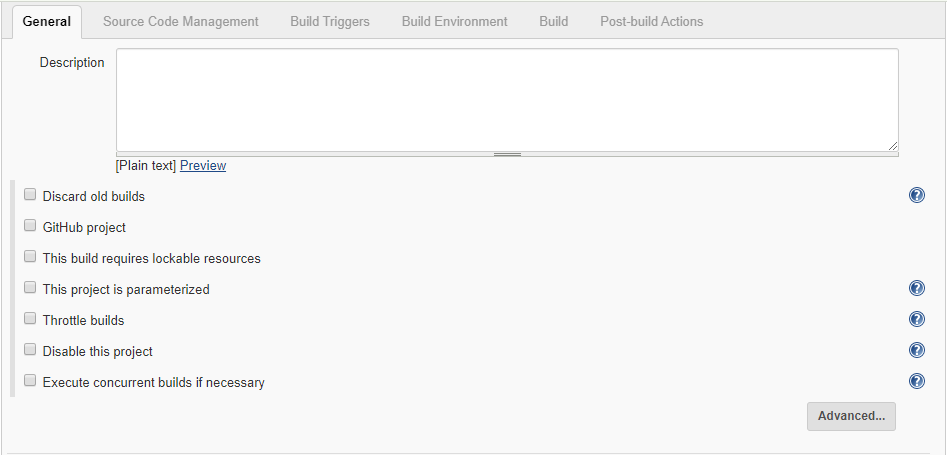
The practice of continuous integration requires that all applicable data be placed under configuration management. Therefore, for a build job developed using the Jenkins CI Framework, the entire ISO 26262 Project must be managed and maintained using a version/revision control system. Once configuration management of the Jenkins build job is in place, you can proceed with creating the complementary Jenkins project.

The following instructions explain how to create a new Jenkins project to launch the Jenkins build job described above. The instructions are written based on the assumption that you are using git for source control.

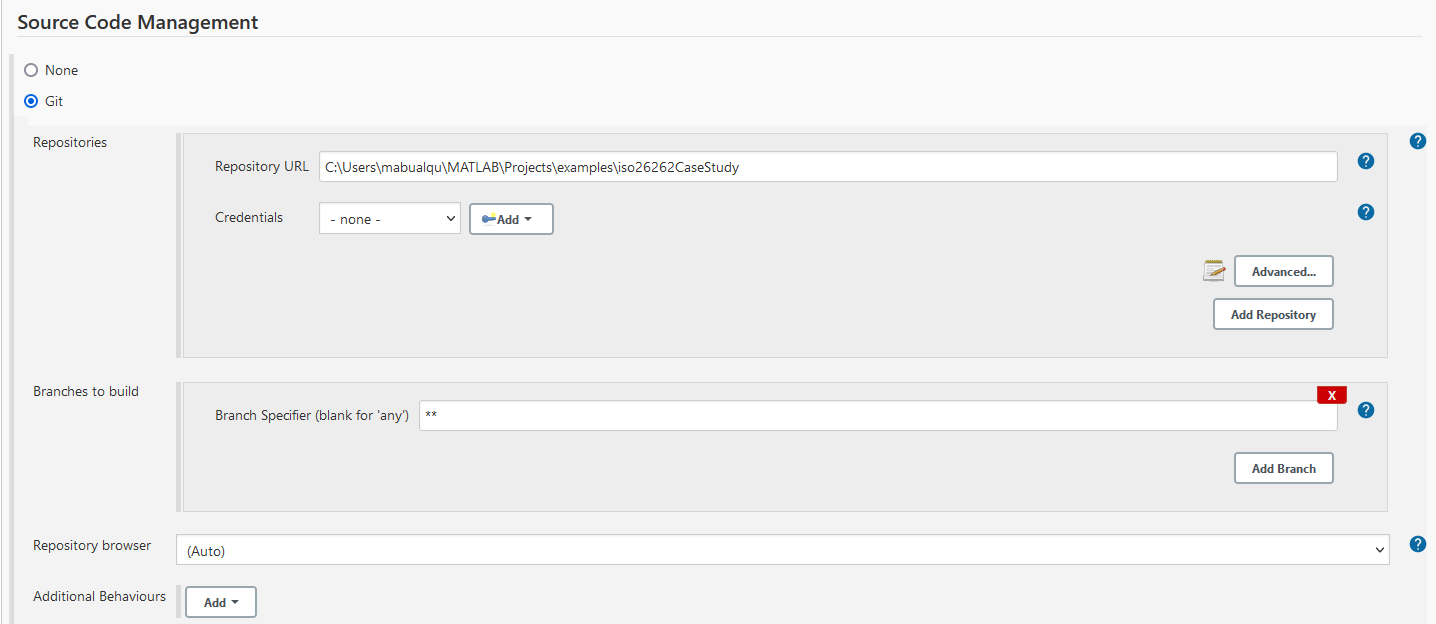
1. On the Jenkins home screen, click *New Item*. This loads the new item screen.
2. On the new item screen, in the *Enter an item name* field, name the new project.
3. Select *Freestyle project*, and then click *OK*. This loads the project configuration screen.



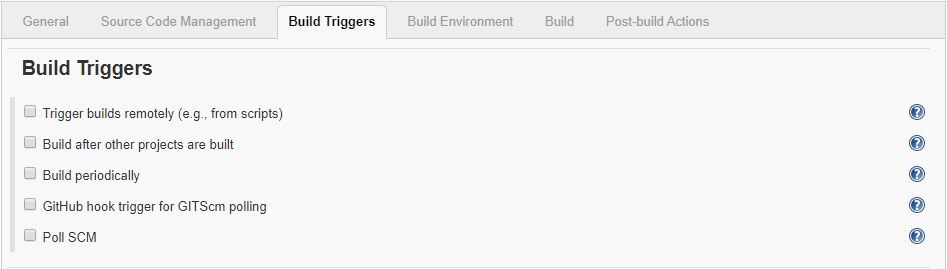
1. Under the *General* tab of the project configuration screen, in the *Description* field, provide a description of the project. Alternatively, you may leave the *General* tab unmodified at its default.



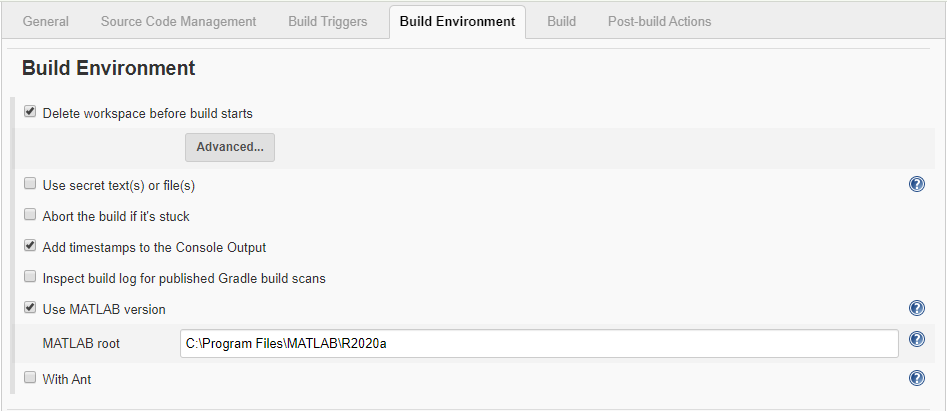
1. Under the *Source Code Management* tab of the project configuration screen, select *git*.
2. In the *Repository URL* field, provide the git repository URL of the ISO 26262 Case Study Project. This should match the full path to the Simulink project root folder.
3. If necessary, use the *Credentials* setting to add user credentials that are required for accessing the git repository.
4. Leave the *Branch Specifier* field as set by default. This setting checks out the project directly into the Jenkins workspace.



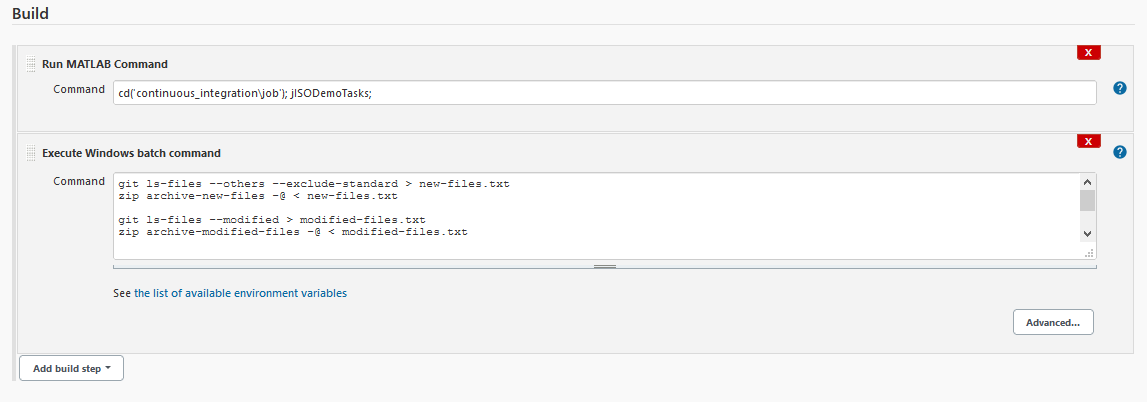
1. Under the *Build Triggers* tab of the project configuration screen, schedule how you want to start the build job. Alternatively, you may leave the *Build Triggers* tab unmodified at its default. Without any triggers, you must start the build job manually.



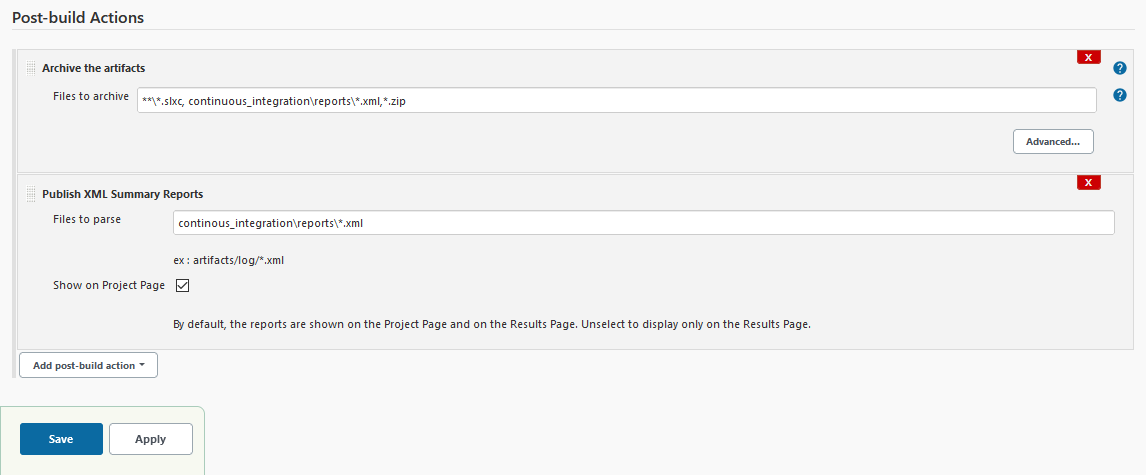
1. Under the *Build Environment* tab of the project configuration screen, select the *Delete workspace before build starts* and *Add timestamps to the Console Output* options.
2. Select the *Use MATLAB version* option, and then in the *MATLAB root* field, provide the full path of the MATLAB installation folder. Use the *matlabroot* function to return the information if necessary.



1. Under the *Build* tab of the project configuration screen, click *Add build step*, and then select *Run MATLAB Command*. This adds a Run MATLAB Command build step.
2. In the *Command* field of the *Run MATLAB Command* panel, enter “*cd('.\continuous\_integration\job'); runJob(ISODemoTasks);*”. Alternatively, you can specify *jISODemoTasks* as a shortcut for *runJob(ISODemoTasks).* Replace *ISODemoTasks* with the name of your Jenkins build job if necessary.
3. Additionally, you can zip all files that are created during the job build and all those modified. The example Windows batch commands below do exactly this.



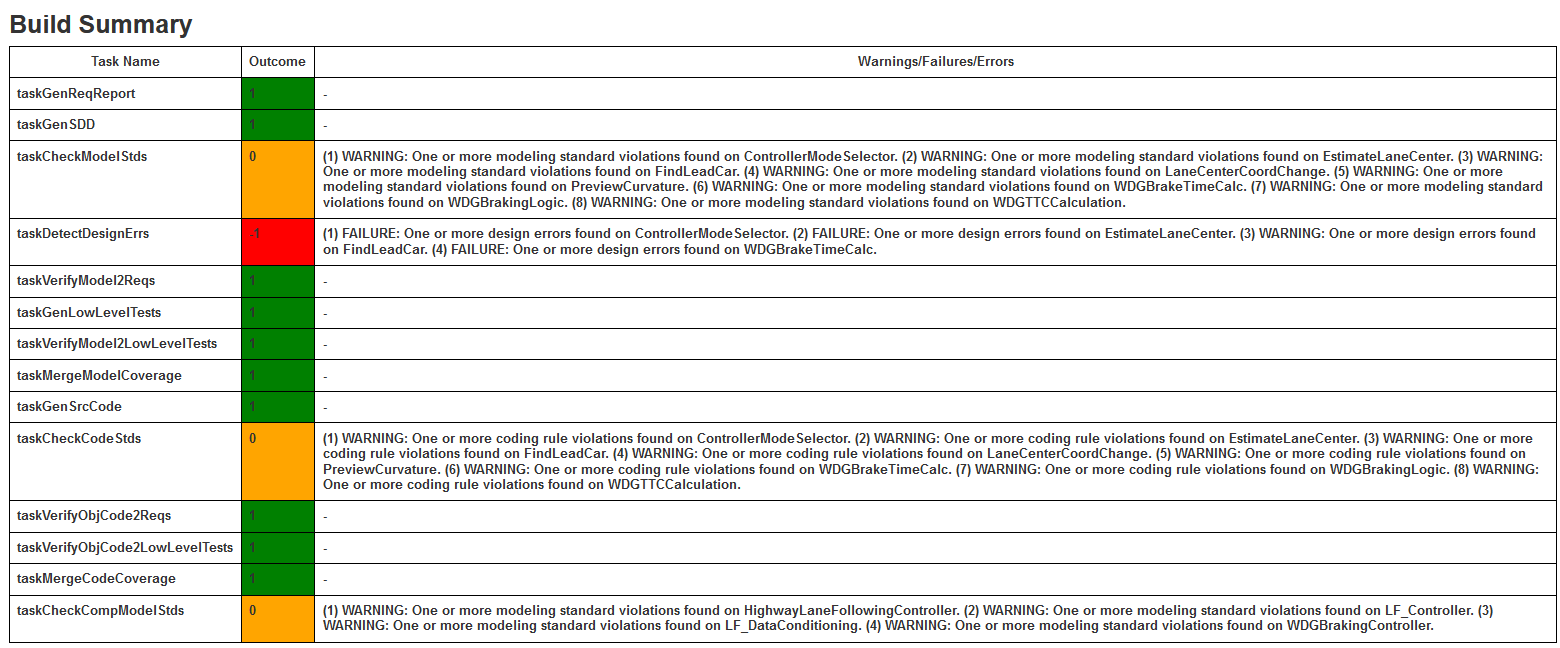
1. Under the *Post-build Actions* tab of the project configuration screen, click *Add post-build action*, and then select *Archive the artifacts*. This adds an Archive the artifacts post-build action.
2. In the *Files to archive* field of the *Archive the artifacts* panel, enter   
   “*\*\*\\*.slxc, continuous\_integration\reports\\*.xml*”,\*.zip.
3. Under the *Post-build Actions* tab of the project configuration screen, click *Add post-build action* again, and then select *Publish XML Summary Reports*. This adds a Publish XML Summary Reports post-build action.
4. In the *Files to parse* field of the *Publish XML Summary Reports* panel, enter “*continuous\_integration\reports\\*.xml*”.



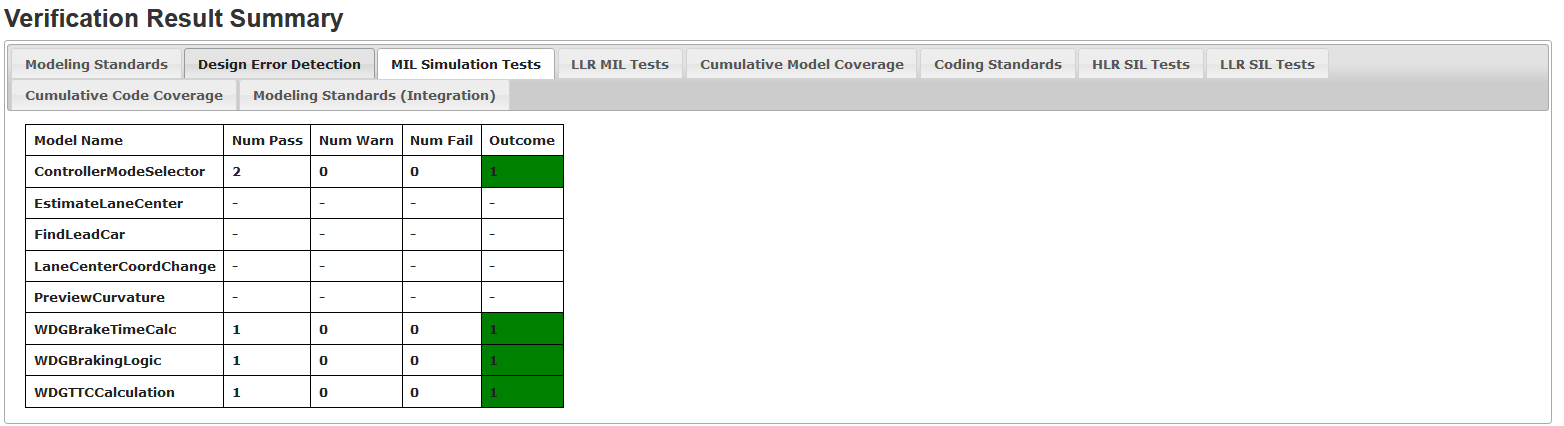
1. Click *Save*. This completes the configuration of the new project and loads the project home screen.

You can now launch the build job from Jenkins. Simply browse to the project you just created in Jenkins, and then click *Build Now*.

Once the build job runs to completion, you can browse to the build in Jenkins to view its published results. Each task of the Jenkins build job reports its overall outcome (PASS/WARN/FAIL) in a table under Build Summary as shown below.



In addition, each software verification task also reports the outcomes (PASS/WARN/FAIL) of individual iterations with itemized results in tabular format under Verification Result Summary as shown below.



# Task Definitions of the Jenkins Build Job

The following table summarizes the task methods available in *ISODemoTasks*:

|  |  |
| --- | --- |
| **Task Method** | **Description** |
| *taskGenReqReport* | Generate a Requirements Report from the technical safety requirements and software safety requirements sets. For example, run  *genReqReport(“TSRS”, [], 'CI')*.  To generate requirements report for the technical safety requirement specification “TSRS”.  The outcome of each iteration is *-1* (FAIL) if the Requirements Report is not successfully generated, *1* (PASS) otherwise.  The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskGenSDD* | Generate an SDD Report from each registered model by calling.   * *genSDD('MODEL', [], 'CI')*.   The outcome of each iteration is *-1* (FAIL) if the SDD Report is not successfully generated, *1* (PASS) otherwise.  The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskCheckModelStds* | Check each registered model for conformance to Software Model Standards by calling   * *checkModelStds('MODEL', [], 'CI')* or * *checkModelStds('MODEL', 'TreatAsTopMdl', 'CI')*.   The outcome of each iteration is determined based on the returned check results, along with an expected report.   * The outcome is *-1* (FAIL) if any check has an error or a failure, *0* (WARN) if any check has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Model Advisor Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskDetectDesignErrs* | Analyze each registered model for design error detection by calling   * *detectDesignErrs('MODEL', [], [], 'CI')*.   The outcome of each iteration is determined based on the returned analysis results, along with an expected report.   * The outcome is *-1* (FAIL) if there is any error or confirmed defect, *0* (WARN) if there is any uncertainty, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Design Error Detection Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskVerifyModel2Reqs* | Verify each registered model against high-level software requirements it implements by calling   * *verifyModel2Reqs('MODEL', [], [], 'CI')* or * *verifyModel2Reqs('MODEL', 'TreatAsTopMdl', [], 'CI')*.   The outcome of each iteration is determined based on the returned verification results, along with two expected reports.   * The outcome is *-1* (FAIL) if any test case has an error or a failure, *0* (WARN) if any test case has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if either the Simulink Test Report or the Model Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskGenLowLevelTests* | Generate low-level tests from each registered model if the coverage of high-level tests is incomplete by calling   * *genLowLevelTests('MODEL', 'CI', true)*.   The outcome of each iteration is *-1* (FAIL) if the test generation report is not successfully generated, *1* (PASS) otherwise.  The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskVerifyModel2LowLevelTests* | Verify each registered model against low-level requirements that are not covered by high-level tests by calling   * *verifyModel2LowLevelTests ('MODEL',[], [], 'CI')* or * *verifyModel2LowLevelTestss('MODEL' , 'TreatAsTopMdl', [], 'CI')*.   The outcome of each iteration is determined based on the returned verification results, along with two expected reports.   * The outcome is *-1* (FAIL) if any test case has an error or a failure, *0* (WARN) if any test case has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if either the Simulink Test Report or the Model Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskMergeModelCoverage* | Assess the overall test coverage for verifying each registered model by calling   * *mergeModelCoverage('MODEL', 'CI')*.   The outcome of each iteration is determined based on the returned coverage results, along with an expected report.   * The outcome is *-1* (FAIL) if there is any error, *0* (WARN) if there is any unexecuted statement, decision, condition, or MCDC, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Code Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskGenSrcCode* | Generate code from each registered model by calling   * *genSrcCode('MODEL')* or * *genSrcCode('MODEL', 'TreatAsTopMdl')*.   The outcome of each iteration is *-1* (FAIL) if the code generation report is not successfully generated, *1* (PASS) otherwise.  The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskCheckCodeStds* | Check the code generated from each registered model for conformance to Software Code Standards by calling   * *checkCodeStds('MODEL', [], 'CI')* or * *checkCodeStds('MODEL', 'TreatAsTopMdl', 'CI')*.   The outcome of each iteration is determined based on the returned check results, along with an expected report.   * The outcome is *-1* (FAIL) if there is any error, *0* (WARN) if there is any MISRA violation, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Bug Finder Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskVerifyObjCode2Reqs* | Verify the code generated from each registered model against high-level software requirements it implements by calling   * *verifyObjCode2Reqs('MODEL', 'SIL', [], [], [], 'CI')* or * *verifyObjCode2Reqs('MODEL', 'SIL', [], 'TreatAsTopMdl', [], 'CI')*.   The outcome of each iteration is determined based on the returned verification results, along with two expected reports.   * The outcome is *-1* (FAIL) if any test case has an error or a failure, *0* (WARN) if any test case has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if either the Simulink Test Report or the Model Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskVerifyObjCode2LowLevelTests* | Verify the code generated from each registered model against low-level software requirements that are not covered by high-level tests by calling   * *verifyObjCode2LowLevelTests('MODEL', 'SIL', [], [], [], 'CI')* or * *verifyObjCode2LowLevelTests('MODEL', 'SIL', [], 'TreatAsTopMdl', [], 'CI')*.   The outcome of each iteration is determined based on the returned verification results, along with two expected reports.   * The outcome is *-1* (FAIL) if any test case has an error or a failure, *0* (WARN) if any test case has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if either the Simulink Test Report or the Model Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskMergeCodeCoverage* | Assess the overall test coverage for verifying the code generated from each registered model by calling   * *mergeCodeCoverage('MODEL', 'SIL', 'CI')*.   The outcome of each iteration is determined based on the returned coverage results, along with an expected report.   * The outcome is *-1* (FAIL) if there is any error, *0* (WARN) if there is any unexecuted statement, decision, condition, or MCDC, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Code Coverage Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |
| *taskCheckCompModelStds* | Check each registered composite model for conformance to Software Model Standards by calling   * *checkModelStds('MODEL', [], 'CI', 'CompositeComponent')* or * *checkModelStds('MODEL', 'TreatAsTopMdl', 'CI','CompositeComponent')*.   The outcome of each iteration is determined based on the returned check results, along with an expected report.   * The outcome is *-1* (FAIL) if any check has an error or a failure, *0* (WARN) if any check has a warning, *1* (PASS) otherwise. * The overridden outcome is *-1* (FAIL) if the Model Advisor Report is not successfully generated, *1* (PASS) otherwise.   The overall execution outcome of the task is governed by the worst outcome of any iteration. |