Huxelerate SDV Performance

Version 1.122.2

Table of Contents

Contents:

SDV performance documentation

Huxelerate Software Defined Vehicle (SDV) performance is a support decision tool to:

1. Correctly size the vehicle E/E architecture by optimizing the number of on-board ECUs and their capabilities
2. Optimize software by design to meet timing, resources and E/E architecture requirements.

The platform leverages on Huxelerate proprietary technology that allows to obtain performance estimates (execution time, resource usage, optimization opportunities) at the single runnable function granularity **without the need of executing the software on the target hardware, considerably saving development, testing and optimization time.**

To provide the estimates of performance, Huxelerate platform provides a library of **architectural models** that modelize the architectural features of a specific processor.

The platform supports multiple development steps, from architecture and communication network definition, to the optimization of the single runnable function that executes on a specific core of a specific processor of an ECU. Thanks to the possibility to create multiple scenarios, it follows software and architecture evolutions, allowing to perform different impact analyses and simulations, while keeping under control timing and performance requirements.

The platform allows to define your vehicle computing architecture in terms of:

- CPU processors inside the ECU
- communication channels

The estimator can:

- Generate Software-In-the-Loop (SIL) wrappers for local simulation of your modules generated via Simulink.
- Provide implementation for I/O interfaces of the autosar runnable, used to provide input data (either randomly generated or loaded from a CSV file), and output data, dumped to a CSV file during the execution.
- Estimate actual performance of your software on your target architectures with information on your runnable function and the functions executed within it.
- Estimate the occupied resources of your functions provided a target execution period.
- Guide you through the optimization process to potentially improve the performance of your software.

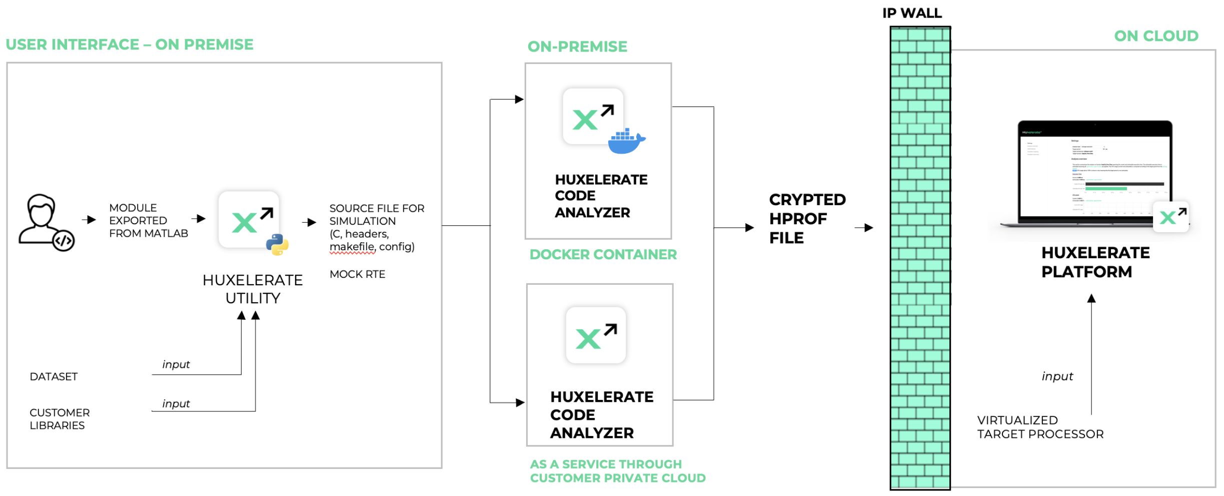
Advantages

The insights provided by the tool allows you to:

- Solve timing issues in advance by understanding actual performance of your modules before testing on hardware
- Automatic verification of timing constraints
- Optimally map variables among available memories analyzing the impact on the overall execution time
- Optimize your code without wasting computing resources

Software Analysis

The analysis of the software can be done at the granularity of the single function/runnable. This means that it is not necessary to have all the source code of the ECU to perform an analysis, nor to compile all the software after code integration.



It is possible to analyze any C/C++ software for which it is available the **source code**. The analysis is performed in three automatic steps:

1. Generation of a wrapper of the target function to run a SIL (Software-In-the-Loop)

2. Static and dynamic code analysis of the target function

3. Generation of the performance report

Step 1. is performed on premise using the workstation of the developer, while step 2. can be either performed on premise (with the support of Docker engine) or as-a-service, using customer's private cloud (i.e. AWS).

Steps 1. and 2. performed in this way guarantee privacy and security of the source code of the customer, that effectively never leaves the infrastructure of the customer.

The output of these 2 phases is a crypted **hprof** file. This file does not contain any source code of the function/runnable, but metrics defining the type of operations, memory accesses and details of the function/runnable like its name, periodicity, and name of interesting functions. Name of functions are employed to provide insight on how to optimize your software.

Once the **hprof** file is generated, it can be uploaded on Huxelerate SDV Platform to obtain a performance report, on multiple target processors.

HPROF file Generation

The generation of the **HPROF** file is handled by Huxelerate Utility, downloadable from Huxelerate Platform, and available both for specific versions of Python, and as an executable for Windows. The utility accepts multiple input flags, depending on the type of exported source code.

Currently, Huxelerate Utility provides support for three categories of source code:

1. Autosar Classic compliant modules

2. Simulink model exported as C code

3. Bare C code from a different source (i.e. handwritten)

For cases 1. and 2. the usage flow starts from exporting the desired module from Simulink. The code needs to be exported as ANSI C code, preferably without specifying any target board. Exported code, together with:

- source libraries

- the mapfile containing the symbols of the module

- optionally, an input CSV file

can be processed by the Huxelerate Utility to generate a SIL wrapper for software execution and a Makefile for compilation and generation of the HPROF file.

Huxelerate Utility

Huxelerate Utility is provided as a Python script, or as an executable for both Windows and Linux. This software is your interface for the generation of the HPROF file. It:

- generates a C program for stimulating the SIL runnables/functions execution for performance analysis purposes (not when used in *bare* mode)
- launch HPROF generation on local machine, or on AWS cloud

The tool reads as input an ARXML file, or an HTML interface definition describing the runnables to test, and generates a corresponding main.c and makefile to compile, instrument and execute the runnables under Huxelerate performance estimator. Optionally, the utility can parse a mapfile to extract the memory mapping of the input/outputs of the runnables and to provide them to Huxelerate performance estimator.

Prerequisites

The executable is self-contained and does not require any additional software to be installed on the machine, however, the Linux executable relies on glibc version 2.27 or higher. Please contact Huxelerate in case you need the utility with a different glibc version.

In case the Huxelerate Utility is downloaded as a python script, the following prerequisites are needed:

- Python (make sure to use the correct version for the downloaded utility)
- Pip
- Huxelerate Autosar library
- Pyarmor
- lxml
- boto3

After installing Python and pip, you can setup your requirements with the requirements.txt file available in this folder

```
python -m pip install -r requirements.txt
```

You are good to go.

HPROF Generation on local machine

The generation of HPROF files on your local machine requires to have Docker engine installed on your local workstation. Docker engine offers OS-level virtualization that allows Huxelerate software to perform the analysis of your software in a containarized environment.

Once Docker engine has been installed on your workstation, it is enough to launch using Huxelerate Utility with default flags and the utility will:

- create a Makefile with a specific version of Huxelerate Performance Estimator image
- download the image of Huxelerate Performance Estimator (only first time)
- launch the analysis leveraging the Docker image
- generate the HPROF file

HPROF Generation as-a-service

The generation of HPROF files using AWS private cloud relies on AWS Lambda. The main infrastructure is composed of 2 S3 buckets (one for the input files, and one for the outputs) and a Lambda function. Everytime new data is uploaded to the S3 input bucket, the lambda is triggered and automatically performs the analysis of provided input, generating the HPROF file in the output bucket. Huxelerate Utility is engineered to completely abstract all the iteration with the cloud service and will automatically:

- Generate the SIL wrapper for the analysis
- Upload the generated files to the input S3 bucket
- Download the result HPROF file from the output S3 bucket

For the specific flags to specify when using HPROF generation as-a-service please refer to the specific section.

It is possible either to install the HPROF generation service on your private AWS cloud, or use the one provided by Huxelerate. In any case, the connection with the service requires authentication either with AWS IAM or using SSO login with your company credentials.

Credentials retrieval via config file (IAM)

Please create the following file:

```
C:\Users\[your-user]\.aws\config
```

Note that the file has no extension.

The file format has to be as follows (more info at https://boto3.amazonaws.com/v1/documentation/api/latest/guide/credentials.html#aws-config-file):

```
[default]
aws_access_key_id=foo
aws_secret_access_key=bar
```

Access Key Id and Secret Access Key have to be populated with the one provided to access the service.

Credential retrieval via SSO Login

Retrieve your credentials using the command (remember to specify a name for the profile when prompted):

```
aws configure sso
```

Store the profile name you have defined in the prompt and use it to set the environment variable *AWS_PROFILE*

```
set AWS_PROFILE=<profile_name>
```

Usage of the tool via a VPC endpoint

The tool offers the possibility to specify a VPC endpoint to use for the connection with a defined bucket. The VPC endpoint can be specified using the *-e* flag, or by setting the environment variable *HUX_AWS_VPC_ENDPOINT_URL*

Huxelerate Utility Usage

For a complete list of flags and functionalities linked to the utility, please refer to the information available by calling the helper function of the utility:

```
python hux-autosar-sil.py -h
```

The Utility provides 4 commands:

- generate_hprof: performs all the steps for the generation of the HPROF file, from the creation of the SIL wrapper archive, to the generation of the HPROF
- generate_hprof_from_archive: performs the generation of the HPROF file starting from a SIL wrapper archive, skipping the generation of the SIL Wrapper

- check_hprof_status: mainly used with as-a-service HPROF generation, checks the status of the generation of an HPROF on the cloud, and in case it is ready it downloads it
- generate_csv_template: generate a csv template with the input stimuli / measures for the simulation

generate_hprof command

This command allows to perform all the steps for the generation of the HPROF file, and in particular:

1. Creates the SIL wrapper archive for the analysis
2. Runs the HPROF generation locally, or on AWS if the specific flag is specified.

The utility supports three different flows:

1. Autosar Classic compliant modules
2. Simulink models (export of HTML report required)
3. Bare C source code

It is possible to specify which flow to follow by specifying:

- *-i* flag for Autosar Classic modules, to specify the either the folder of the ARXML files, or the file itself
- *-is* flag for Simulink models, to specify the *<module_name>_interface.html* file
- *–bare* flag for bare C source code

For all flows, the utility requires to specify the sources for the analysis by means of .c and .h files. The first can be specified with the *-c* flag, while the second using the *-I* flag. Both flags can be specified multiple times. These two flags are used by the util to create a Makefile, with the specific include paths and source files to compile and analyze the software.

To stimulate the software with specific dataset/measures, it is possible to specify a CSV file containing the inputs with *-v* flag. Note that this flag is note defined with *–bare* flag. For instruction on the file format, please refer to this section.

The run of this command generates the following outputs:

- a zip file containing all the generated files (mock RTE) for the SIL
- the HPROF files of the runnables/software analyzed

Both outputs can be organized using the flags *-z* to specify a name for the zip file, and *-ho* to specify a name for the HPROF.

Analyses can be carried on locally, or on AWS as described in the specific section

By default, the generate_hprof command performs both SIL wrapper archive generation, and the generation of the HPROF file. It is however possible to generate only the SIL wrapper archive, specifying the *-g* flag. The utility will terminate the processing once the archive has been generated, to allow code inspection before HPROF generation. It is then possible to generate the hprof using the generate_hprof_from_archive command

generate_hprof_from_archive command

This command generated the HPROF file starting from a SIL wrapper archive, skipping the creation of the SIL itself. It can be used to re-generate an HPROF file, starting from a previously created SIL wrapper archive, or as a subsequent command of the generate_hprof command with the *-g* flag specified. The command is standard for each flow, and accept HPROF generation on AWS.

An example command to generate an hprof in *output_hprof_fullpath* starting from a SIL wrapper called *<input_sil_wrapper>* and with generation on AWS is:

```
python hux-autosar-sil.py generate_hprof_from_archive -z
<input_sil_wrapper> -ho <output_hprof_fullpath> -a -b <bucket_name>
```

check_hprof_status command

This command can be used when an asynchronous processing has been triggered, to check if the generation of an HPROF file on AWS has completed. If the file is available, the command will automatically download it on your local machine.

```
python hux-autosar-sil.py check_hprof_status -z <input_sil_wrapper> -ho
<output_hprof_fullpath> -b <bucket_name>
```

The example command above, is used to check the generation of the HPROF file that - if successfully generated - will be downloaded to *output_hprof_fullpath*. The input sil wrapper is *<input_sil_wrapper>*.

generate_csv_template command

This command generates a CSV file template, containing the input stimuli for the simulation. The template can be populated by the user with specific measures to analyze the runnables/functions in a real-case scenario, to get a more insightful analysis.

See the section in the specific flow to see example commands to generate the CSV template.

Local vs cloud generation of HPROF files

By default, the generation of HPROF file is performed on the local workstation, and requires to have Docker engine installed. In the example above, the analysis is performed leveraging a proprietary cloud service available on AWS cloud. The flags that specify this behavior are *--aws* and *-b*. The flag *--aws* is to specify the HPROF generation on AWS cloud, while the *-b* flag is used to specify the name of the bucket to use for the generation of the HPROF on AWS. The two flags need to be defined together if the analysis is performed in the cloud. The name of the bucket can be also defined using the *HUX_AWS_BUCKET_NAME* environment variable.

From a performance perspective, the generation of HPROFs is based on AWS lambda, that can be invoked cuncurrently exploiting the parallelism offered by AWS Lambda infrastructure.

When the HPROF generation on AWS is chosen, the utility automatically uploads the zip file containing the sources, and waits for the generation of the HPROF file, or for a failure. When the HPROF file is generated, it is automatically downloaded by the utility to the local workstation. It is possible to launch the AWS Lambda process asynchronously by specifying the *-y* flag and then use the *check-hprof-status* command to check the processing status and download the HPROF file at a later stage.

HPROF generation scenarios

Autosar Classic compliant modules

The analysis of Autosar Classic compliant software, relies on the ARXML files describing the Software Component of the runnable. The ARXML files need to be provided to the utility to get information such as:

- Runnables name
- Runnables periodicity
- Port interfaces
- Input/Output types
- etc ...

These details are used to generate the SIL (Software In the Loop) wrapper (including a mock RTE) for the compilation, including example CSV files that can be generated to provide specific measures (datasets) to stimulate the execution of the runnable.

HPROF generation command for Autosar compliant software

By default, Autosar compliant software provided is analyzed at the SWC (Software Component) level. The utility simulates a schedule of the runnables in the SWC, hence the measures provided as input need to refer to the input ports of the SWC itself. As a result, you can expect multiple HPROF files, one for each runnable in the SWC.

Here's an example command to generate the HPROF performing the analysis at the SWC level on AWS:

```
python hux-autosar-sil.py generate_hprof \
  -i <arxml_files_folder> \
  -m <map_file> \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  -v <csv_files> \
  --aws \
  -z <sil_archive_name> \
  -b <aws_bucket_name> \
  -ho <hprof_output_file>
```

The *-i* flag is used to provide the ARXML files to process. In this case the utility will process all the ARXMLs in *<arxml_files_folder>* folder, to identify the components, runnables and variables, and process the map file *<map_file>* to identify variable mapping.

The *-I* and *-c* flags are necessary for the creation of the Makefile, to know which are the include paths to use and the source files to compile respectively.

The *-v* flag is used to specify a CSV file containing the input dataset. If not specified, a random dataset will be generated to test the runnable.

This command produces the following outputs:

- a zip file containing all the generated files (mock RTE) for the SIL
- the HPROF files of the runnables analyzed

To organize the outputs produced by the command, it is possible to use the *-z* flag to specify the name of the zip file, and the *-ho* flag to specify the base name of the HPROF files.

Analysis at runnable level vs SWC level

The default behavior of the utility for Autosar compliant software is an analysis at SWC component level. It is however possible to:

1. perform an analysis at SWC level but generate the HPROF file for a single runnable

2. perform an analysis at runnable level

The main difference between the two analyses is in the input measure provided to the tool. In case 1., as the analysis is at the software component level, the utility expects the input signals of the SWC. Here's an example command:

```
python hux-autosar-sil.py generate_hprof \
  -r <runnable_name> \
  -i <arxml_files_folder> \
  -m <map_file> \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  -v <csv_file_of_swc> \
  --aws \
  -z <sil_archive_name> \
  -b <aws_bucket_name> \
  -ho <hprof_output_file>
```

Notice the *-r* flag used to specify the name of the runnable to analyze.

In case 2. the utility expects to have the input signals of the runnable. As in this example:

```
python hux-autosar-sil.py generate_hprof \
  -rs \
  -r <runnable_name> \
  -i <arxml_files_folder> \
  -m <map_file> \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  -v <csv_file_of_runnable> \
  --aws \
  -z <sil_archive_name> \
  -b <aws_bucket_name> \
  -ho <hprof_output_file>
```

The utility generates a single HPROF file for the *<runnable_name>* to be stimulated with the input CSV file containing input measures of the runnable.

Use a mapfile to provide symbol location

To get a more accurate performance analysis, the analysis can include information regarding the memory mapping of the I/Os of the runnables and the location of the code in the flash memories. These information can be parsed by the utility during the analysis, by providing a mapfile with the *-m* flag.

By providing the mapfile, the performance report takes into consideration the location of the symbols, and, based on their exact location, is able to provide a more accurate estimate for a specific architecture When the software is then analyzed on a different target, the symbol location will not be taken into consideration.

Generate a CSV template file for the input stimuli

Depending on the level of analysis performed (runnable vs SWC level), it is possible to generate a template CSV file, containing the name of the input stimuli that are necessary to perform the analysis with a real-case dataset. In case the input CSV file is not provided, the input stimuli are randomly generated.

If you want to provide stimuli for the function to be simulated, you must also submit a CSV file organized as in the example below. The *-v* flag is used to specify the input CSV file. Each column of the CSV file needs to have the variable name as first entry (header) and all the values for each timeframe as entry below. If the variable name is composite, then its name must be described as *<parent variable name>.<child variable name>* as reported in the example.

Example file for analysis at runnable level:

```
Rte_Var_1_P,Rte_Var_2_P.var_2_C
0.0,228.0
0.001,228.0
0.002,228.0
```

When the analysis is performed at SWC level, the variables provided must be the ones in input to the SWC. Furthermore, there is an additional field represented by the **time** keyword, expressed in seconds.

Example file for analysis at runnable level:

```
time, Rte_Var_1_Inport,Rte_Var_2_Inport.var_2_C
0.1,0.0,228.0
0.2,0.001,228.0
0.3,0.002,228.0
```

The following command allows you to generate a CSV template file containing all the variables useful for simulating a SWC.

```
python hux-autosar-sil.py generate_template -i <arxml_files_folder> -o
<output_csv_file_name>
```

When the analysis focuses on a single runnable, it is necessary to use the *-rs* and *-r* (omittable in case of a single runnable in the SWC), to specify the analysis at runnable level, and the specific runnable for which the utility needs to generate the template:

```
python hux-autosar-sil.py generate_template -rs -r <runnable_name> -i
<arxml_files_folder> -o <output_csv_file_name>
```

The *-i* flag specifies the folder and/or the filename of the ARXML files to process. The generated *<custom_name>.csv* is the output. .. note:

```
Generating the template using this command is strongly recommended to
avoid possible errors.
```

Avoid RTE stub creation

When providing the ARXML file, the utility will automatically create an RTE stub for dynamic code analysis. It is possible to avoid the RTE stub creation as follows:

1. use the *–skip-rte-stub-generation* so that the utility does not automatically generate the stub and
2. specify the new sources that can be used for the RTE stub with the *-I* and *-c* flags.

Simulink model exported as C code

In the Simulink model flow, to leverage on the automation offered by Huxelerate Utility, it is necessary to export the module from Simulink as C code, **including the generation of the HTML report**. The report called '<component_name>_interface.html' is in fact used to retrieve information about:

- step functions name
- step functions periodicity
- input/outputs type
- etc...

These details are used to generate the SIL wrapper (including a mock RTE) for the compilation, including example CSV files that can be generated to provide specific measures (datasets) to stimulate the execution of the step function.

HPROF generation command for Simulink exported models:

The utility fetches the information regarding the step functions from the HTML report, that is possible to generate during the generation of the source code. In particular, the required HTML file is the one with information about the interfaces, following the name pattern *"<module_name>_interface.html"*.

```
python hux-autosar-sil.py generate_hprof \
  -is <html_module_interface> \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  -v <csv_files> \
  --aws \
  -z <sil_archive_name> \
  -b <aws_bucket_name> \
  -ho <hprof_output_file>
```

The HTML report is specified with the *-is* flag, while the other flags works as in the Autosar compliant scenario: *-c* and *-I* are used to specify source files, *-v* specifies the input measures and the *--aws* and *-b* flags are used for the generation of HPROF on AWS (see `AWS Hprof Generation`_)

Generate a CSV template for the input stimuli

If you want to provide stimuli for the function to be simulated, you must also submit a csv file organized as in the example below. The *-v* flag is used to specify a CSV file as input. Each column of the CSV file needs to have the variable name as first entry (header) followed by all the values, one for each timeframe. If the variable name is composite, then its name must be described as *<parent variable name>.<child variable name>* as reported in the example.

Example file for the analysis of the step function:

```
rtU.inport1,rtU.inport2
0.0,228.0
0.001,228.0
0.002,228.0
```

Template generation command:

```
python hux-autosar-sil.py generate_template \
  -is <html_module_interface> \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  -o <output_csv_file_name>
```

The *-is* flag specifies the filename of the HTML module interface file to process. It is also necessary to specify the include paths and the source files required by the runnable via *-I* and *-c* flags. These are needed to let the utility infer the variables and data structures necessary for input stimulation.

The generated *<custom_name>.csv* is the output.

Note

Generating the template using this command is strongly recommended to avoid possible errors.

Bare C code

This category allows to analyze any C source code without any support file (i.e. ARXML or HTML report). The main differences with the other categories is that the developer is responsible for:

- wrapping the function to analyze, with a main function that gets the inputs, provide them to the function and reads the output
- specify the function to analyze
- specify the periodicity of the function

HPROF generation command for bare C code

For the bare C code example, the utility expects to have all the source files available for the compilation specified with *-c* and *-I* flags, and will not generate any RTE stub.

```
python hux-autosar-sil.py generate_hprof --bare \
  -r <function_to_analyze> \
  -p 10 \
  -I <library_folder_to_include> \
  -I <library_folder_2_to_include> \
  -c <library_folder_to_include> \
  -c <library_folder_2_to_include> \
  --aws \
  -z <sil_archive_name> \
  -b <aws_bucket_name> \
  -ho <hprof_output_file>
```

It is mandatory to specify the name of the function to analyze with the *-r* flag. Furthermore, it is possible to specify the periodicity of the function to analyze, expressed in milliseconds, with the *-p* flag.

Additionally, it is possible to specify additional compile flags with the *-q* flag. These flags will be used during the compilation step.

The generated *<custom_name>.hprof* is the output of the performance analysis. Upload the report to Huxelerate Platform to get the final report.

SDV Performance SaaS Platform

SDV Performance platform is a cloud platform that allows you to create a digital representation of the computing resources available in your vehicle, and to analyze and optimize the performance of your software on different scenarios of applications.

With SDV Performance platform, you can:

- estimate the performance of your software on a multitude of hardware architectures, without re-compiling your software, skipping the integration phase and without any hardware resource
- get performance insights and optimization suggestions, without actually executing on the physical hardware
- analyze the interactions of different software components to monitor your resources and timing requirements
- identify timing issues months before, saving time and resources

User types

The platform is engineered to support three categories of users:

1. Admin
2. Normal
3. Guest

Admin and **Normal** users are a category of users that belong to the same organization. The main difference between the two, is that **Admin** users can see, and operate on the Organization Settings, with details on the current active license and other sensitive information, and are responsible to invite/remove active users from the platform. **Admin** are by default assigned to each created project, and will see all the projects in homepage.

Guest user category is specific for collaborators that operate **outside** the organization. An example use for this category is for software providers, that are not willing to share the source code of the applications with the organization. With this category of user, they have the possibility to generate HPROF files on their local machine, and upload it to a specific project in the platform, without sharing the source code of the appplication. In this way:

- users of the organization can have a complete view on the performance of the software
- guest users can monitor the performance of the software that are developing for the customer

The view for this category of users is limited to the performance reports, but actions (like delete) can be taken only on the performance reports originated from HPROF files uploaded by the guest user.

Organization Settings

From this section, it is possible to monitor information regarding:

- Status of Licenses
- Active Users under the organization
- Active projects with details on assigned users for each project.

The access to this page is limited to **Admin** users, that have also the functionality to add/remove new users to the organization.

Settings

This section allows each user to see and modify details and configurations linked with its account. From here, it is also possible to generate the REST api token, to be used to:

- interact with the platform using the REST apis
- validate licenses with products linked to Huxelerate Platform

Note

The token allows authentication to your account without your username and password. Misuse or loss of the token, may open your personal account to a third party, that could perform operations on your behalf. To this end, we suggest to keep the token safely stored, and generate tokens with a defined expiration date.

Support Platform

For any issue linked to the platform or Huxelerate utility, in this section it is possible to access to a support platform to get support.

Projects details

Each user has the possibility to create **projects**, and to share them with other colleagues. For each project there are two category of users:

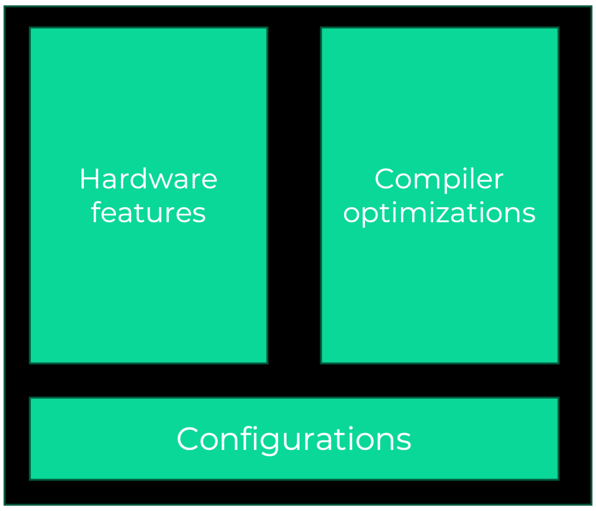
- **project manager** : the creator of the project - can share the project with other users, tag resources with protected tags and operate on all resources
- **developer** : a contributor to the project - can see available analyses and create new, but cannot operate on the analyses of others

Inside a project there are three sections:

1. ECU Types: specify the ECUs available in your project
2. System Definition: Specify the hardware architecture, software components, task and runnable mapping and timing requirements for the analysis
3. Analyses: get performance insights at runnable/function or scenario level

1. ECU Types

The virtualization of the ECU offered by Huxelerate is at the processor level. For each processor, Huxelerate develops, and offers in the platform an **architectural model** of the processor that can be interpreted as a digital twin of the physical processor, that operates as it physical counterpart. The model allows to get multiple insight on your software on a target architecture from a performance point of view.



The model offered in the platform is the result of two ingredients:

1. hardware features (such as number and type of processing elements available on the target architecture, or the bandwidth/latency of the available memories)
2. compiler optimizations (such as the compilation strategy)

Each model is configurable, so that it is possible to test the architecture under different circumstances (i.e. cache on/cache off). The main configuration of the model involves:

- Clock frequency of the components
- Memories (i.e. cache enabled in specific segments, prefetchers, etc...)
- Compilation flags
- Compilator used

The choice of a specific architectural model and its configuration, defines an **ECU Type**. In this section, it is possible to instantiate multiple ECU Types, that will be used for the analysis in the project. For an analysis to be performed on a target architecture, it is necessary that the relative ECU type is defined in this section. Having multiple ECU Types is very useful, especially when comparing the performance of a software on different targets, even if the target is the same architecture, but configured differently. Once an ECU Type has been defined in the project, it is then possible to instantiate it inside a scenario for analysis, or use it to get a performance report.

2. System Definition

System definition sections collects all the information necessary to perform a **scenario analysis**. A scenario is defined as a photography, real or hypotetical, of the physical system, on which we want to test the performance of the system. It is composed of:

- one or more ECUs, that can be interconnected with a communication channel (network)
- a definition of network data (usually defined via DBC file)
- a definition of the software components available on the ECUs
- a set of Timing contstraints to be checked during the simulation
- a Task and runnable mapping on the cores of the available ECUs

It is possible to define multiple scenarios, and copy the definition of the system from another scenario, so that it is possible to **track the evolution of the system**

3. Analyses

There are three categories of analyses that is possible to perform, depending on the license available:

1. Runnable Performance: analyze the performance of your software at the runnable level. Obtain current and attainable execution time, functions breakdown, CPU usage on a multitude of hardware processors and optimization suggestions
2. Network Analysis: analyze network load and response time in isolation, identify scenarios that may originate high network load

3. Resource Analysis: get current ECU usage of a set of software running on your ECU

4. Timing Analysis: simulate the behavior of your software on a set of target ECUs connected by a network, to monitor resource usage, constraints violation and software interactions.

Platform sections

- System Definition
- Analyses

System Definition

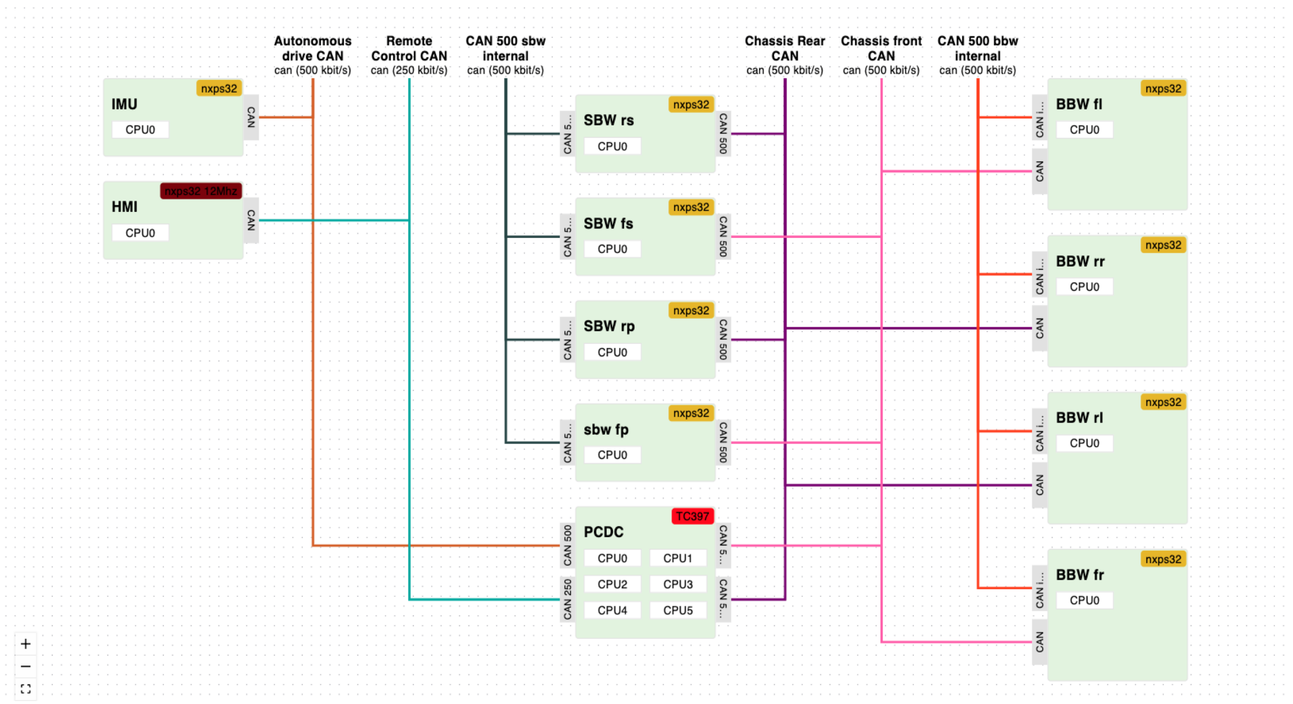
The definition of the system allows to perform a simulation of the entire system taking into consideration the interactions among runnables, ECUs and messages on the network. During the simulation it is possible to check a set of timing constraints to validate the functionality of the overall system.

The platform allows to create multiple **scenarios** to track the evolution of the architecture, and simulate different use cases that would be very expensive to test on the real hardware architecture. Scenarios can be created from scratch, or evolve from an another scenario and allow to:

- define the overall architecture by means of **ECU types** (defined in the ECU Types section) and **communication channels** (i.e. CAN network with specific bandwidth)
- specify data for each specific **network**
- define the **software components** available in the system
- insert a set of **timing constraints** to be validated during the simulation
- define multiple tasks and **task mappings** to be simulated

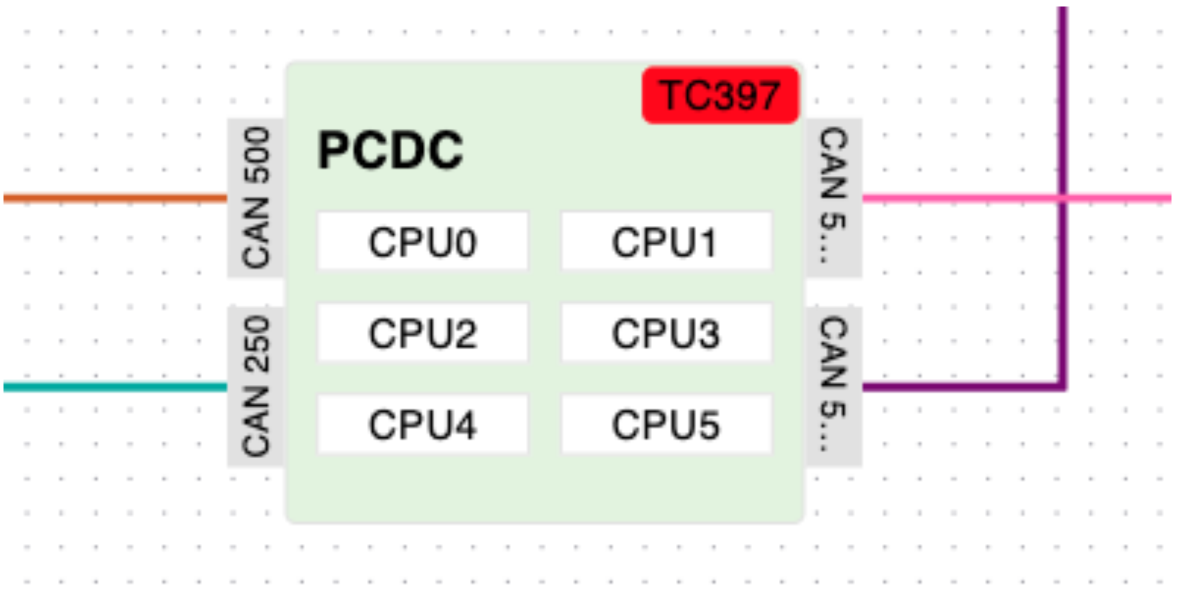
Scenarios are pretty powerful, as they allow to test, in a digital environment, your system under different conditions. As an example, it is possible to simulate the current situation of the architecture to check for timing issues, check the impact of the addition of a new piece of software into an existing architecture, or simulate the behavior of the software with a different set of tasks, or mapping of tasks on the different cores of a single ECU or on a different ECU.

ECUs



ECUs section allows to create a digital representation of the ECUs available in your architecture, and how they interconnect with specific networks. In this section it is possible to instantiate multiple ECUs, starting from the ECU Types defined in the specific section. For each instantiation, it is possible to define:

- a name for the ECU Type instance
- the type of the of the ECU as in the specific section
- a cost for the ECU - this is useful to obtain a total cost for the simulated solution
- the network interfaces of the ECU. For each interface it is possible to define a name, and the type of interface.



Above you can find how it looks like an ECU called PCDC, that implements an ECU Type that we have defined as TC397 and contains 4 network interfaces of type CAN.

Network Data

To obtain a detailed simulation of the system, that includes an analysis of the traffic on specific networks, it is possible to specify the **Network Frames** of each specific network, defined in the ECUs section. A Network Frame is defined by:

- Name
- Period
- Size of Payload
- Network on which it passes
- [optional] Identifier
- [optional] wether it uses or not the extended format

The platform will simulate the traffic on the network based on the information provided in this section, that can be defined manually, or automatically, uploading a specific **DBC** file. The information inserted in this section, can be also used to analyze the network as a standalone element, by using the Network Analysis functionality

Software Components

This section hosts the information regarding the software available in the scenario. Software is defined by:

- **Software Components** that can be composed by one or more runnables (or step functions)
- communication **interfaces**

Software Components contain information about runnables (or step functions), and for each runnable:

- **Assigned Ports** identified by the name, the type of access (i.e. read/write), port type (i.e. provide/require) and the relative interface.
- **Events** that identify how the function is called (i.e. periodically with a specific period)
- **Operation Modes** that define when the runnable is active during the simulation/execution (see section on Operation Modes)

Each element defined in this section, can be univocally identified by two elements, the **UUID** and the **Entity Reference**. Both this information are used by the platform to identify the component, and to understand the connections among different components. The manual modification of an entity reference, may cause the platform to lose track of a relationship between a runnable and a specific port. For more information on how UUID and Entity Reference are used please see the section Consideration on UUID and Entity Reference

The population of this specific section can be time consuming, and evolves while software is being developed. For this reason, we offer three methodologies to populate it:

1. specifying one, or more, ARXML files
2. using the “SWCs from Hprofs” button
3. manually

If compliant with the **Autosar** specification, it is possible to directly specify the ARXMLs containing the information of the Software Component, and the platform will automatically populate all the data, keeping the relationships among elements defined with UUIDs and Entity References. The platform will also check for the consistency of the data provided by the ARXML, so that it is possible to identify accidental error or manual modification to the ARXML.

In case the ARXML is not available, it is possible to use the **SWCs from Hprofs** button. This button fetches the information on the Software Components directly from the HPROF uploaded in the Runnable Performance section. Note that this data could be incomplete, depending on the information stored in the HPROF upon generation.

Operation Modes

Operation Modes allow to simulate specific use cases of the system (i.e. key turn). In this section it is possible to define a set of **states** in which the system could be, and the relatives runnables' state for the specific system state. Once a set of states is defined, it is possible to define the activation or deactivation of a specific state in the Simulation Schedule

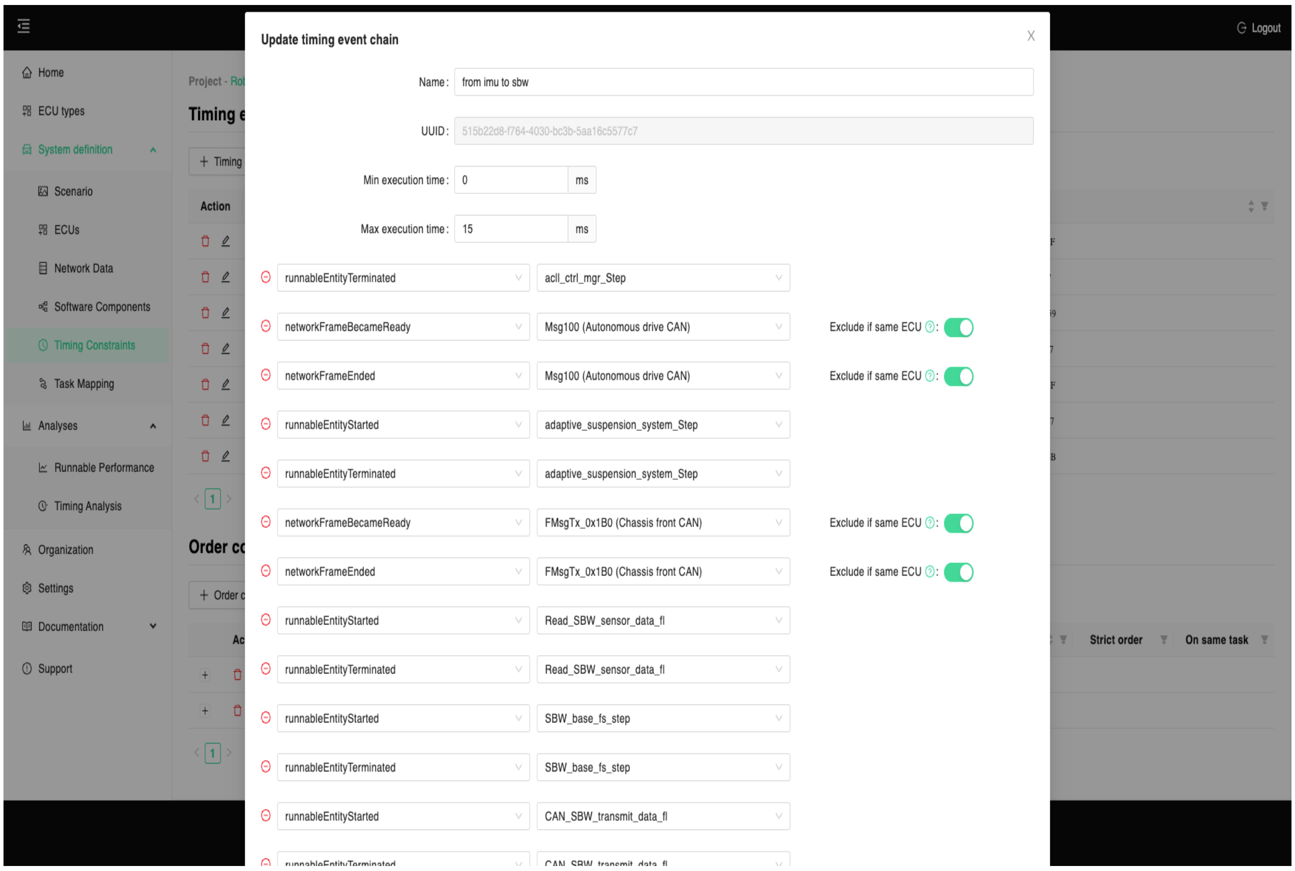
Timing Constraints

While the simulation itself provides information regarding performance (i.e. execution time) and resource occupancy, it is possible to place additional checks related to timing and interactions of runnables/functions and messages passing on the networks. It is possible to define two categories of timing constraints:

1. Timing event chains

2. Order constraints

Timing event chains allows the definition of a maximum time budget for a specific set of events, for which it has been defined an order of execution. The event is defined either by an operation on a runnable/function, such as start/end, or by an operation on a network frame, such as its start, end or when it became ready.



Update timing event chain

Name:	from imu to sbw
UUID:	515b22d8-f764-4030-bc3b-5aa16c5577c7
Min execution time:	0 ms
Max execution time:	15 ms

runnableEntityTerminated	acll_ctrl_mgr_Step	
networkFrameBecameReady	Msg100 (Autonomous drive CAN)	Exclude if same ECU: ✓
networkFrameEnded	Msg100 (Autonomous drive CAN)	Exclude if same ECU: ✓
runnableEntityStarted	adaptive_suspension_system_Step	
runnableEntityTerminated	adaptive_suspension_system_Step	
networkFrameBecameReady	FMsgTx_0x1B0 (Chassis front CAN)	Exclude if same ECU: ✓
networkFrameEnded	FMsgTx_0x1B0 (Chassis front CAN)	Exclude if same ECU: ✓
runnableEntityStarted	Read_SBW_sensor_data_fl	
runnableEntityTerminated	Read_SBW_sensor_data_fl	
runnableEntityStarted	SBW_base_fs_step	
runnableEntityTerminated	SBW_base_fs_step	
runnableEntityStarted	CAN_SBW_transmit_data_fl	
runnableEntityTerminated	CAN_SBW_transmit_data_fl	

In the example above, it is shown how to define an event chain that involves both runnable/functions and messages on specific buses with a maximum time budget defined of 15ms. When a simulation is launched, the system will automatically add this check in the analysis, and will raise an error in case there is one or more instances of this event chain that is failing. With the platform is easy to relocate runnables/functions or messages on different ECUs or networks. To this end, the "Exclude if same ECU" flag permit to avoid to communication on the network in case the two runnables/functions that produce/consume the message are located in the same ECU.

Event chains are pretty powerful as can be used in multiple scenarios. They can be used to monitor specific runnable/function-message interaction, or to check the periodicity of specific runnables. In general, thanks to event chains it is possible to monitor any interaction between events that need to happen within a time budget.

The second type of timing constraints is represented by **order constraints**. These constraints allow to define the expected order of execution of runnables within tasks. This constraint is not evaluated during simulation, but rather during task mapping, and will not be considered for runnables/functions mapped to different tasks.The modal accepts an ordered list of runnables/functions and permits the definition of two flags. The first one checks for a strict ordering, validating that no runnable/function is executed between the defined order; the second one validates that the runnables/functions execute on the same task.

Task Mapping

The section allows the creation of multiple task mappings and, as for the scenario, it is possible to copy and evolve the configuration of a specific task mapping. For each specific mapping it is possible to define:

- Tasks
- mapping of runnable/function to tasks
- mapping of task to ECU and core of the ECU

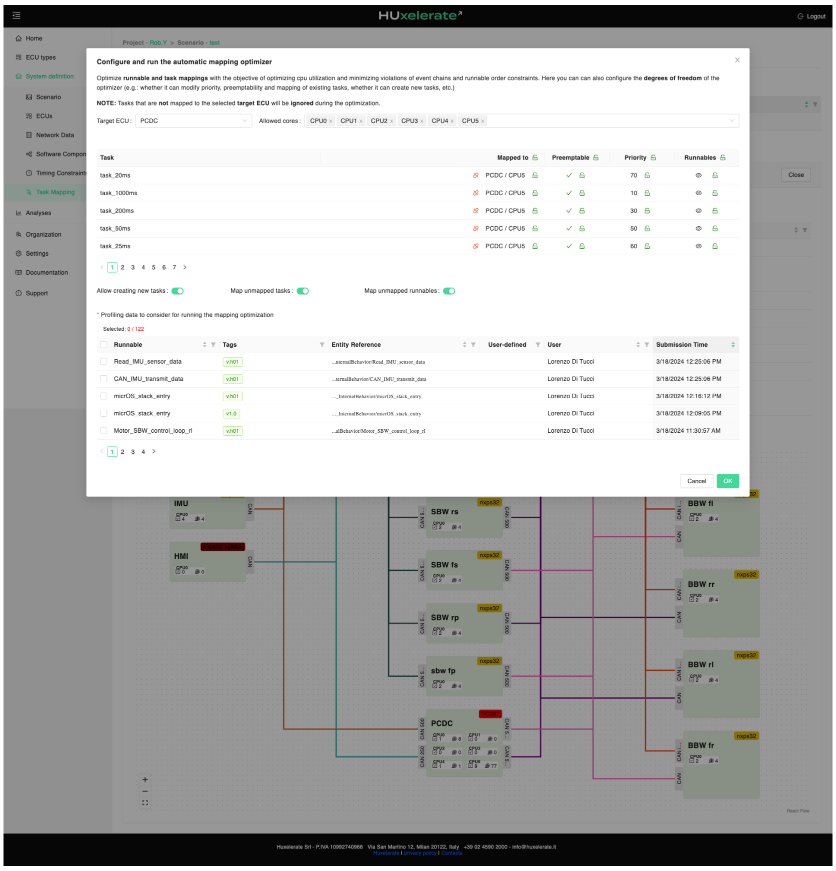
Tasks are identified by a name, an optional UUID and the Entity Reference that is automatically generated when task is manually created. For each task it is necessary to provide the trigger (i.e. a periodic task with its specific period), the mapping of the task on a core of a specific ECU, its priority and preemptability. For each task then, the system automatically filters runnables/functions with a compliant period that could be mapped to it. At this stage, the system automatically checks the order constraints defined in the dedicated section.

As for other sections of the platform, there are three ways to perform a task mapping:

- Providing the Os, Rte and EcuC ARXML files
- Using the **automatic mapper**
- manually

Define a mapping using the Automatic Mapper

The automatic mapper is a utility that performs a design space exploration (DSE) to suggest a possible solution to the mapping problem. The output of the utility is a suggestion of mapping of periodic tasks and runnables/functions on the target architecture.



Configure and run the automatic mapping optimizer

Optimize **runnable and task mappings** with the objective of optimizing cpu utilization and minimizing violations of event chains and runnable order constraints. Here you can can also configure the **degrees of freedom** of the optimizer (e.g.: whether it can modify priority, preemptability and mapping of existing tasks, whether it can create new tasks, etc.)

NOTE: Tasks that are **not** mapped to the selected **target ECU** will be **ignored** during the optimization.

Target ECU: PCDC   Allowed cores: CPU0 CPU1 CPU2 CPU3 CPU4 CPU5

Task	Mapped to	Preemptable	Priority	Runnables
task_20ms	PCDC / CPU5	✓	70	
task_1000ms	PCDC / CPU5	✓	10	
task_200ms	PCDC / CPU5	✓	30	
task_50ms	PCDC / CPU5	✓	50	
task_25ms	PCDC / CPU5	✓	60	

1 2 3 4 5 6 7

Allow creating new tasks: ● Map unmapped tasks: ● Map unmapped runnables: ●

* Profiling data to consider for running the mapping optimization

Selected: 0 / 122

Runnable	Tags	Entity Reference	User-defined	User	Submission Time
Read_IMU_sensor_data	v.h01	...nternalBehavior/Read_IMU_sensor_data		Lorenzo Di Tucci	3/18/2024 12:25:06 PM
CAN_IMU_transmit_data	v.h01	...ternalBehavior/CAN_IMU_transmit_data		Lorenzo Di Tucci	3/18/2024 12:25:06 PM
micrOS_stack_entry	v.h01	..._InternalBehavior/micrOS_stack_entry		Lorenzo Di Tucci	3/18/2024 12:16:12 PM
micrOS_stack_entry	v1.0	..._InternalBehavior/micrOS_stack_entry		Lorenzo Di Tucci	3/18/2024 12:09:05 PM
Motor_SBW_control_loop_rl	v.h01	...alBehavior/Motor_SBW_control_loop_rl		Lorenzo Di Tucci	3/18/2024 11:30:57 AM

1 2 3 4

It is possible to launch the mapping optimizer either on an empty mapping (i.e. no tasks have been defined yet), or with an intermediate/full set of tasks defined. It is possible to define multiple degrees of freedom to allow the mapping optimizer to:

- permit the creation of new periodic tasks
- allow the mapping of unmapped periodic tasks on ECU cores
- allow the mapping of unmapped periodic runnables on ECU cores
- define the cores on which the automatic mapper is allowed to operate

In case tasks are already defined in the mapping, it is possible, with task-granularity, to avoid changes by the automatic mapper to:

- task mapping
- preemptability
- priority
- runnables mapped to the task

In its default configuration, the automatic mapping optimizer will have the capacity to create new periodic tasks, choose a specific mapping, preemptability, priority and set of periodic runnables to map onto it. The automatic mapping optimizer will apply a strategy to minimize resource occupancy while avoiding runnables/functions execution overlap. The final result of the automatic mapping optimizer, will be a mapping proposal, that needs to be accepted or refused by the user. The report will highlight the main modification with current mapping to highlight differences.

The utility is pretty useful, especially when evaluating a different architecture with respect to the one defined at the beginning, and a task mapping has not been defined yet; or to optimize an existing mapping.

UUID and Entity Rerefence

UUIDs and Entity References are used to univocally identify resources in a project. Thanks to this relationship, the platform is able to track the evolution of the single components, and to correctly simulate the interaction among different components (i.e. two runnables and their respective ports during a timing simulation). It can happen however, that this references are modified during the development stages. In this case, the platform has in place some checks to make sure that components are not defined multiple times, but with different ids. It is always up to the developer, to make sure that the references are constantly updated, so that the simulation can be as precise as possible.

Analyses

This section hosts an overview of all the analysis categories performed inside the platform. Analyses differ from the type and granularity of the analysed component. The basic block for the data available in the platform is the analysis of the runnable/function. Starting from the performance details available in this analysis, it is possible to obtain the subsequent analyses.

Currently supported analyses categories are:

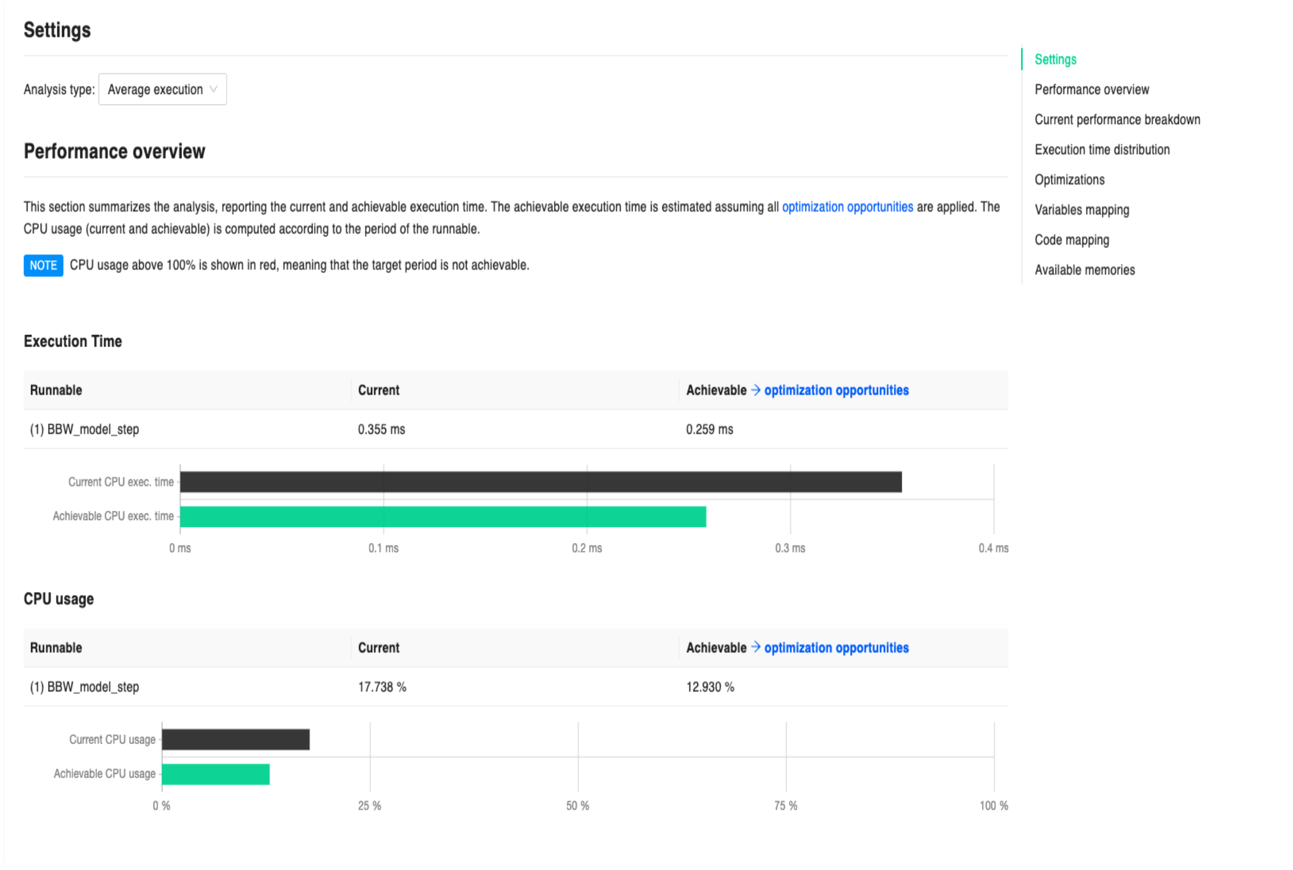
- Runnable Performance
- Network
- Resource Analysis
- Timing Analysis

Runnable Performance

The starting point for the analysis of a piece of software is the generation of its HPROF file. HPROF files can be uploaded in this section of the platform (both with REST API or manually) to obtain a performance report.

Each HPROF file can produce multiple performance analyses, depending on the number of ECU Type defined. For each ECU type, it is possible to obtain an analysis on a specific core of the ECU. Each entry can be classified with specific tags, that help filtering the desired HPROFs while performing specific analyses.

A performance report is an interactive document composed of multiple sections that provide performance information of a software targeting a specific ECU type. The analysis can be obtained simulating an **average** or **longest** execution. The analysis is performed by stimulating the software multiple times, depending on the provided input. The first one will provide average numbers based on the dynamic analysis performed stimulating the code with the provided dataset during HPROF file generation; while the second one will provide the analysis obtained with the dataset originating the longest execution time.



Settings

Analysis type: Average execution

Performance overview

This section summarizes the analysis, reporting the current and achievable execution time. The achievable execution time is estimated assuming all optimization opportunities are applied. The CPU usage (current and achievable) is computed according to the period of the runnable.

NOTE CPU usage above 100% is shown in red, meaning that the target period is not achievable.

Execution Time

Runnable	Current	Achievable → optimization opportunities
(1) BBW_model_step	0.355 ms	0.259 ms

CPU usage

Runnable	Current	Achievable → optimization opportunities
(1) BBW_model_step	17.738 %	12.930 %

Performance Overview

The section summarizes the analysis, reporting the current (in black) and achievable (in green) execution time. Achievable execution time is estimated assuming that all the Optimization opportunities are applied to the code. Data regarding CPU usage (both current and achievable) is computed according to the period of the runnable either specified by hand or collected by the Huxelerate utility during the HPORF generation.

Whenever CPU usage is above 100%, data is shown in red, meaning that the target period is not achievable.

Code coverage and cyclomatic complexity are also provided in this section, to give an overview of the code complexity and the amount of code executed during the analysis. If multiple runnables are under analysis, respective bar graphs are also provided to compare the metrics for different runnables.

Current Performance Breakdown

The performance breakdown provides information about the execution time and CPU usage for each function involved in the execution of the runnable. For each function, the report provides data about both execution time and cpu occupancy, both in a bar chart and an interactive table that can be used to order the functions. Note that the metrics refer to self-time / self-usage, not including the metrics of the called functions.

Execution Time Distribution

In this section it is possible to analyze the execution time of each data point / measure provided in input for the analysis. A chart shows the current and achievable execution time distribution, aggregated by number of executions. In the chart it is easy to identify if there are specific outliers depending on the input code. More detailed data is provided in a tabular fashion. The table provides information at the granularity of the single input / function call. Developers can order the table to identify which is the speicific dataset causing an increase (or decrease) in the execution time, and act accordingly.

Optimizations

Optimizations opportunities identified by the tool are shown in this section. Available optimizations are shown in green, while not applicable, or already applied ones, are greyed out. The total saving provided by the optimization opportunities, assume that the optimization is applied to all functions involved in the execution of the main function.

Optimization opportunities section is pretty powerful, as it allows to see the impact of a specific optimization **before** applying it. In this way, it is possible to focus on the optimizations that have the greatest impact. In these regards, the tool is useful to perform impact analyses: the final decision is always of the developer/user, that knows if the specific optimization can be applied, or not.

Optimization focus on:

- code optimization
- variables mapping optimization

- code mapping optimization

Variables and code mapping optimization, can be better targeted when providing a **map** file during HPROF generation. In this way, the tool knows the exact position of the symbols. It is also possible to manually define the position of the symbols, by specifying them on the JSON file provided to the Huxelerate utility during HPORF generation.

Variables mapping

This section shows the mapping of global variables to their corresponding memory location. By default, all global variables are mapped to a specific memory location that depends on the target ECU type. To test the impact of using different memory locations, it is possible to modify the variables mapping by specifying the variable addresses in the configuration file provided during the instrumentation step of the Huxelerate utility. The available memories and their corresponding address spaces are listed in the available memories section.

Code Mapping

This section shows the mapping of functions' code to their memory location. By default, all functions are mapped to a specific memory location that depends on the target ECU type. To test the impact of using different memory locations, it is possible to modify the functions mapping by specifying the function address in the configuration file provided during the instrumentation step of the Huxelerate utility. The available memories and their corresponding address spaces are listed in the available memories section.

Available memories

List of available memories and corresponding address spaces for the target architecture.

User-defined HPROF

Whenever it is not possible to automatically generate an HPROF file using the Huxelerate utility, it is possible to manually insert an entry using the dedicated button at the top of the page. To create an user-defined HPROF file, the runnable must be defined in the software components section . Once the runnable and scenario are selected, it is possible to define the average and maximum execution time, and the its' standard deviation.

Network Analysis

Network Analysis functionality can be used to analyze network load and response time in isolation (before or without ECU software availability), and can be used to:

- obtain the bus load
- analyze worst-case jitter and response time at the network message granularity

- validate network quality KPIs such as maximum deviation from scheduled message period
- statistically simulate the periodicity of sporadic messages and stress test the network under different use-case scenarios.

Launch a Network Analysis

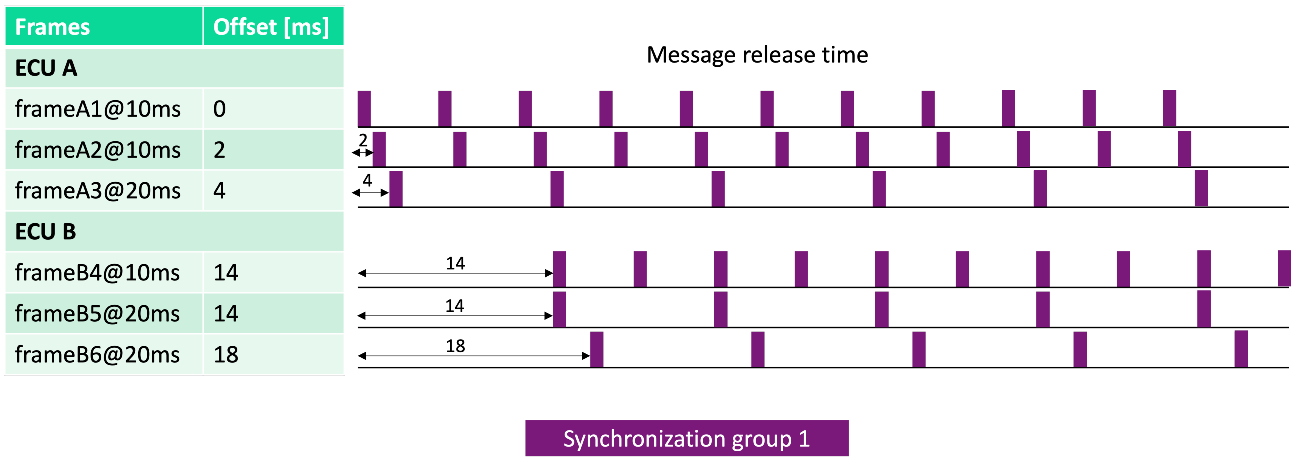
By clicking on the “New Network Analysis” button, it is possible to customize the configuration of a new Network Analysis: * you must set the name for the analysis, the scenario and network to be analyzed; * moreover, you can specify a global release jitter, describing the variability with which outgoing messages are released from the transmission queue, whether to “Synchronize ECUs” and “Synchronize different message periods” start instants, as explained in the following section.

Periodic messages’ synchronization options

We categorize periodic network messages into different synchronization groups. Messages in the same group are scheduled to request transmission together.

- Enabling Inter-ECU synchronization (“Synchronize ECUs” switch), ECUs are assumed to enable the communication stack in the exact same instant, leading to messages having the same period **across all the ECUs** to fall in the same synchronization group.
- Enabling Intra-ECU synchronization (“Synchronize different message periods” switch), each ECU’s communication stack is assumed to start scheduling messages of every period at the same instant, leading to all the messages of each sender ECU to fall in the same synchronization group, ignoring their period.

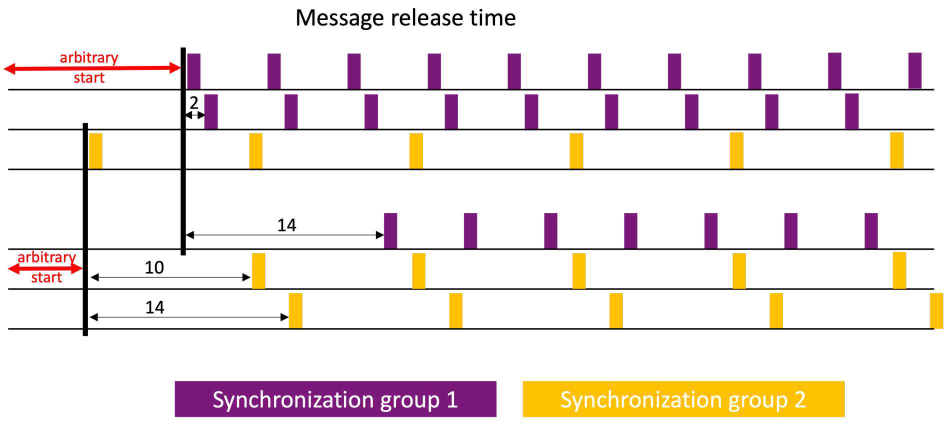
As an example, consider two ECUs with 3 messages each, each of them with a specific offset. Enabling both “Synchronize ECUs” and “Synchronize different message periods” results as follows:



Frames	Offset [ms]
ECU A	
frameA1@10ms	0
frameA2@10ms	2
frameA3@20ms	4
ECU B	
frameB4@10ms	14
frameB5@20ms	14
frameB6@20ms	18

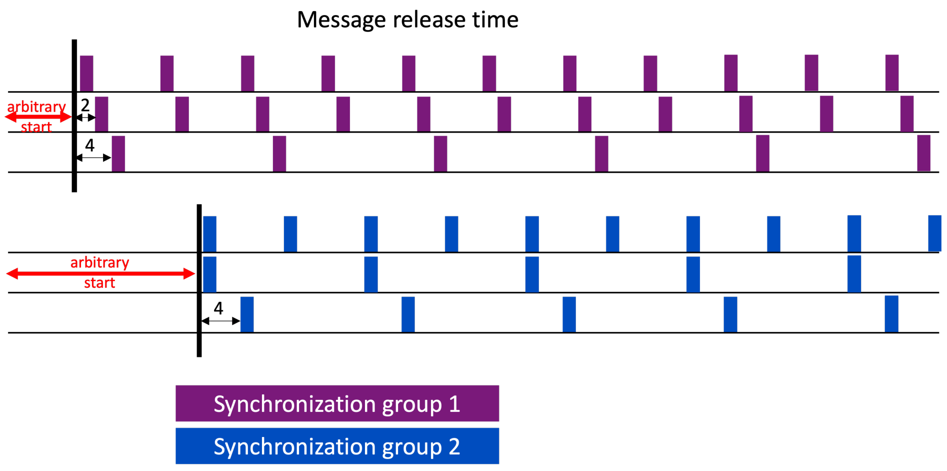
Enabling only “Synchronize ECUs”, the example results as follows:

Frames	Offset [ms]
ECU A	
frameA1@10ms	0
frameA2@10ms	2
frameA3@20ms	4
ECU B	
frameB4@10ms	14
frameB5@20ms	14
frameB6@20ms	18



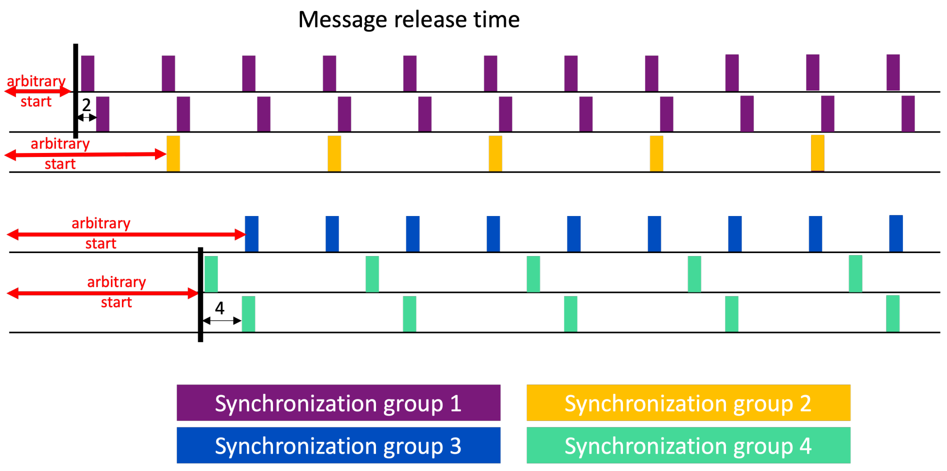
Enabling only “Synchronize different message periods”, the example results as follows:

Frames	Offset [ms]
ECU A	
frameA1@10ms	0
frameA2@10ms	2
frameA3@20ms	4
ECU B	
frameB4@10ms	14
frameB5@20ms	14
frameB6@20ms	18



Disabling both “Synchronize ECUs” and “Synchronize different message periods”, the example results as follows:

Frames	Offset [ms]
ECU A	
frameA1@10ms	0
frameA2@10ms	2
frameA3@20ms	4
ECU B	
frameB4@10ms	14
frameB5@20ms	14
frameB6@20ms	18



Analysis

Once a network analysis has been launched and the results are ready, it is possible to navigate through results thanks to an interactive visualization, identifying:

- bus load
- misconfigurations such as overlapping signals within frames or signals extending beyond frame length
- potential performance issues such as message periodicity inversions
- underutilized frames

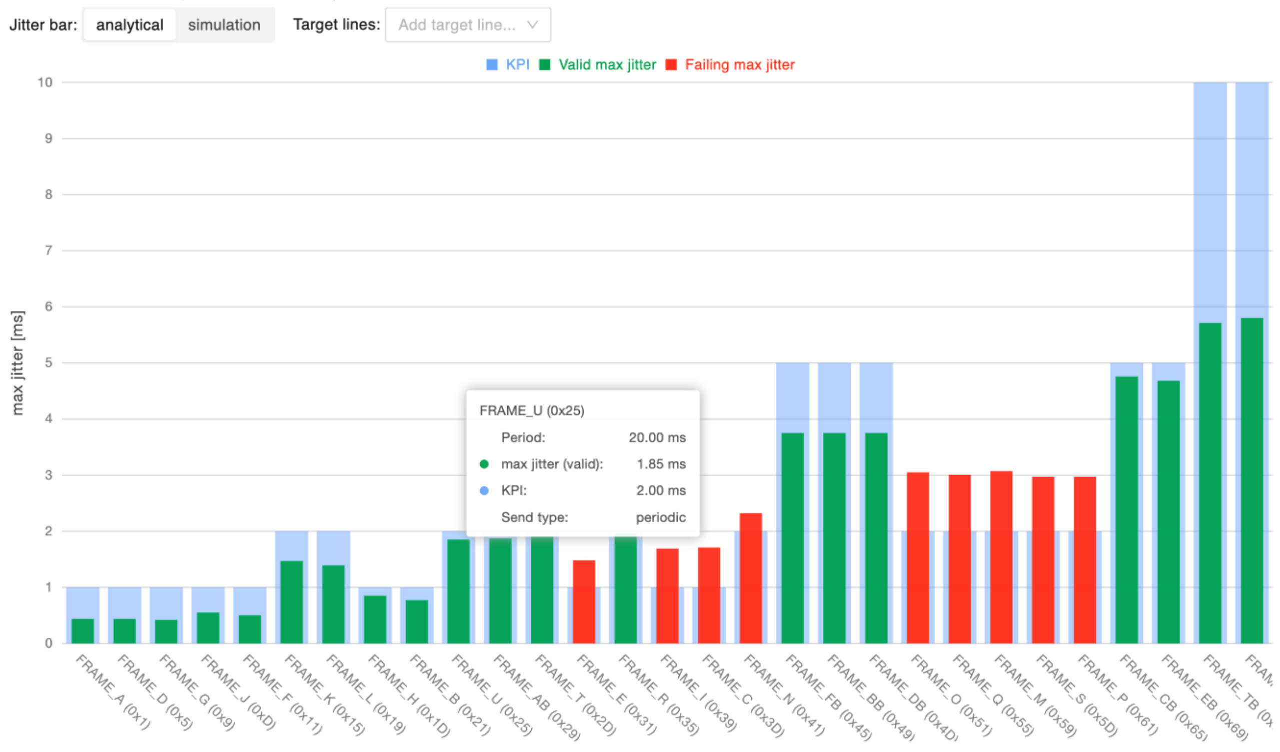
The interactive report allows to filter out messages by period or by type (periodic, sporadic, mixed) to focus on the specific messages and provide messages in two charts.

Moreover, the performance of the network can be evaluated according to two KPIs: Jitter and Worst-Case Response Time (WCRT), detailed in the following sections.

Jitter analysis

By **jitter** of a network message, we refer to a KPI accounting for the variability in the period between two consecutive periodic transmissions. For example, if a message is supposed to be sent every 10ms (i.e. the message period), and its recorded transmission times are 0.3ms, 10.4ms, and 21.2ms, the average jitter is 0.45ms and the maximum jitter is 0.8ms. The report highlights the maximum jitter for each message and allows to set a boundary to the maximum acceptable jitter, as a percentage of the message period.

Frame-wise jitter analysis


Jitter bar: analytical simulation

Target lines: Add target line...

FRAME_U (0x25)

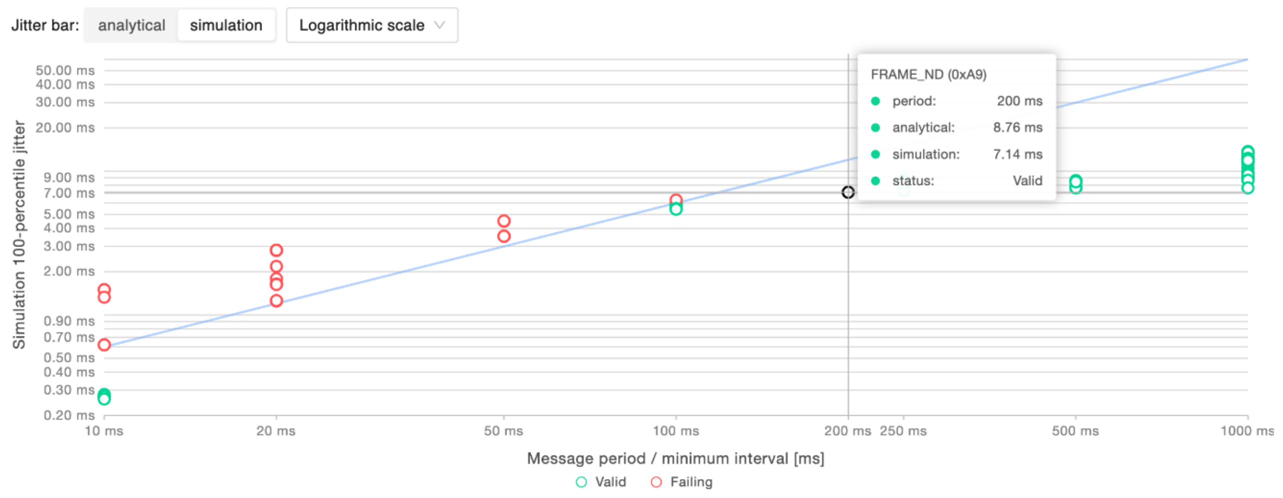
	Period:	20.00 ms
●	max jitter (valid):	1.85 ms
●	KPI:	2.00 ms
	Send type:	periodic

This chart shows the maximum jitter for each message, allowing to identify messages that experience excessive jitter, which may reduce the reliability of the network. On the x-axis of the chart there are increasing message identifiers (lower priority last), while the y-axis provides the maximum jitter time in ms. By setting a boundary for the maximum acceptable jitter (as a percentage of the message period), messages within the boundary are represented with a green bar, messages violating it with a red bar.

The maximum jitter is available both according to the analytical, worst-case, model and using the statistical simulation's results, toggling the "Jitter bar" selector. Independently from the selected jitter computation method, it is possible to overlay the chart with target lines highlighting: * the maximum jitter according to the analytical model * the maximum jitter according to the simulation results, at a user-defined percentile over the distribution of the simulated jitter values.

Finally, clicking on a bar will open the relative message inspection.

Period-wise jitter analysis



This chart allows to validate the reliability of the network in terms of stability for each message period. In the chart, every point corresponds to a message. On the x-axis there are increasing message periods (or minimum transmission intervals for non-periodic messages), while the y-axis provides the maximum jitter time in ms. The set boundary for the maximum acceptable jitter is represented by a line that separates messages within such a boundary, represented as green points, from messages violating it, represented as red points. In order to better accommodate a wide range of jitter values, it is possible to choose the type of visualization scale (linear or logarithmic). In the example above, information is reported in logarithmic scale, and the maximum jitter is set to be 6% of the period of the messages.

The maximum jitter is available both according to the analytical, worst-case, model and using the statistical simulation's results, toggling the "Jitter bar" selector.

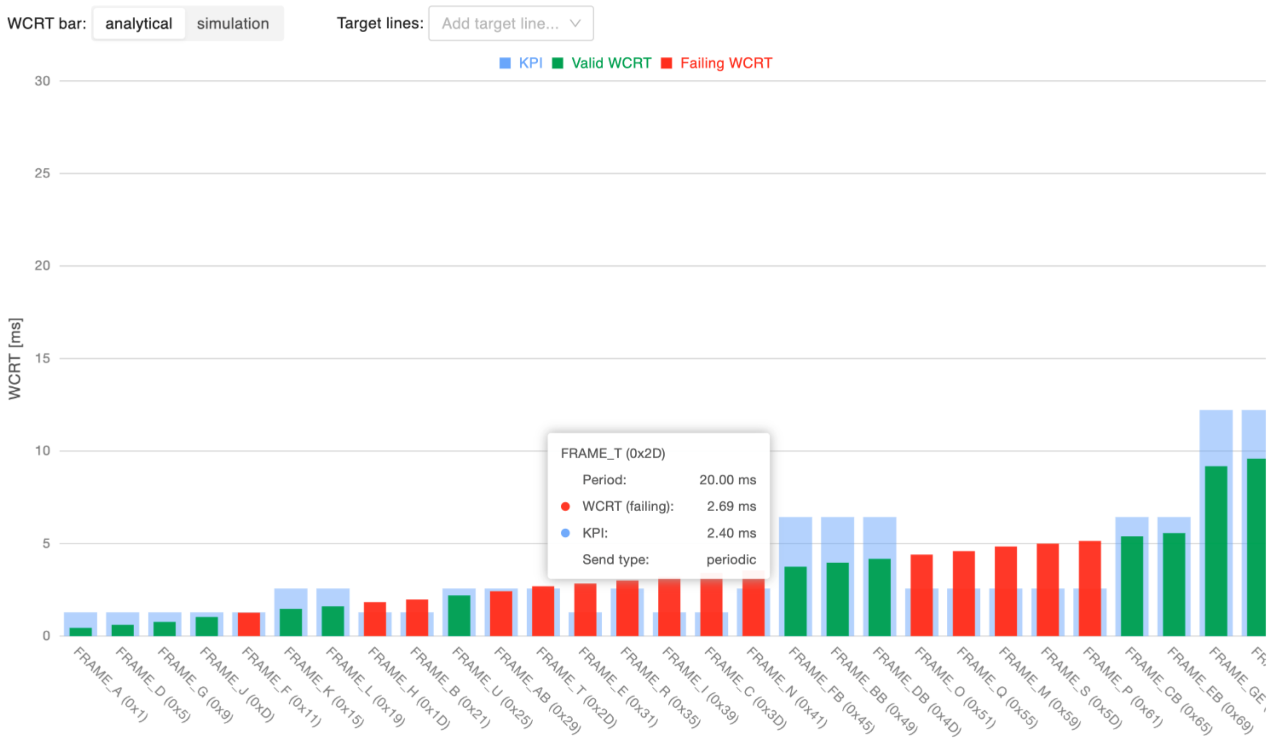
Hovering on a specific message it is possible to get its details: its period, its analytical maximum jitter and the simulated maximum jitter at the specified percentile.

Worst-Case Response Time (WCRT) analysis

By **worst-case response time (WCRT)** of a network message, we refer to a KPI accounting for the maximum time a message takes to be transmitted from the sender to the receiver, accounting for: * Queuing delay: the time a message spends in the transmission queue before being sent * Transmission time: the time it takes to send the message over the network * Release jitter: the variability in the time the message is released from the transmission queue

The report highlights the WCRT for each message and allows to set a boundary to the maximum acceptable response time, as a percentage of the message period (or minimum transmission interval for non-periodic messages).

Frame-wise WCRT analysis


WCRT bar: analytical simulation

Target lines: Add target line...

FRAME_T (0x2D)

Period: 20.00 ms

WCRT (failing): 2.69 ms

KPI: 2.40 ms

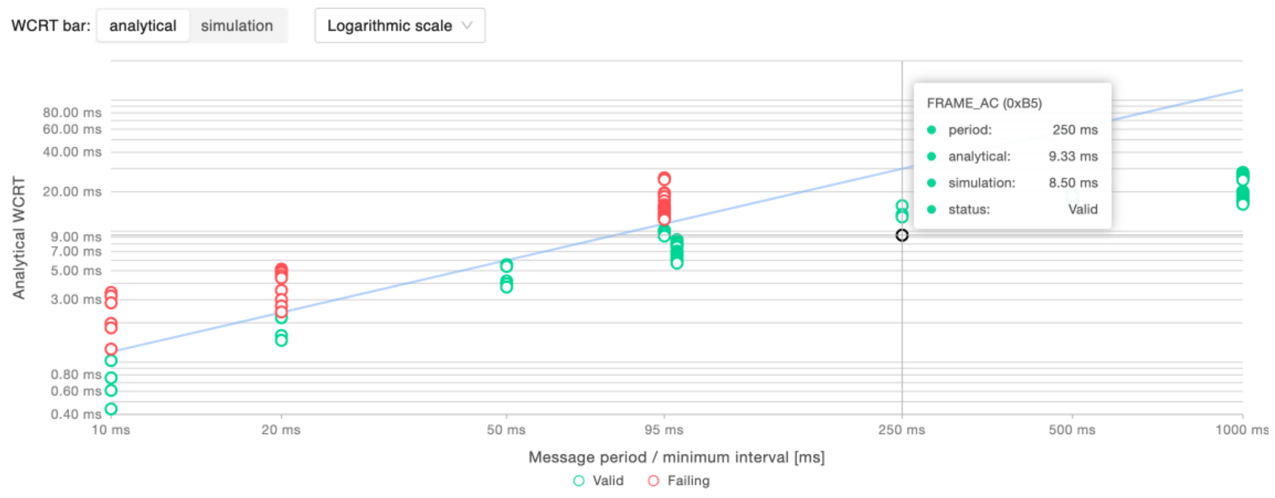
Send type: periodic

This chart shows the WCRT for each message, allowing to identify messages that experience excessive response time, which may reduce the reliability of the network and result in potential actual/expected message periodicity inversions. On the x-axis of the chart there are increasing message identifiers (lower priority last), while the y-axis provides the WCRT time in ms. By setting a boundary for the maximum acceptable WCRT (as a percentage of the message period, or minimum transmission interval, for non-periodic messages), messages within the boundary are represented with a green bar, messages violating it with a red bar.

The WCRT is available both according to the analytical model and using the statistical simulation's results, toggling the "WCRT bar" selector. Independently from the selected WCRT computation method, it is possible to overlay the chart with target lines highlighting: * the WCRT according to the analytical model * the WCRT according to the simulation results, at a user-defined percentile over the distribution of the simulated WCRT values.

Finally, clicking on a bar will open the relative message inspection.

Period-wise WCRT analysis



WCRT bar: analytical simulation

Logarithmic scale

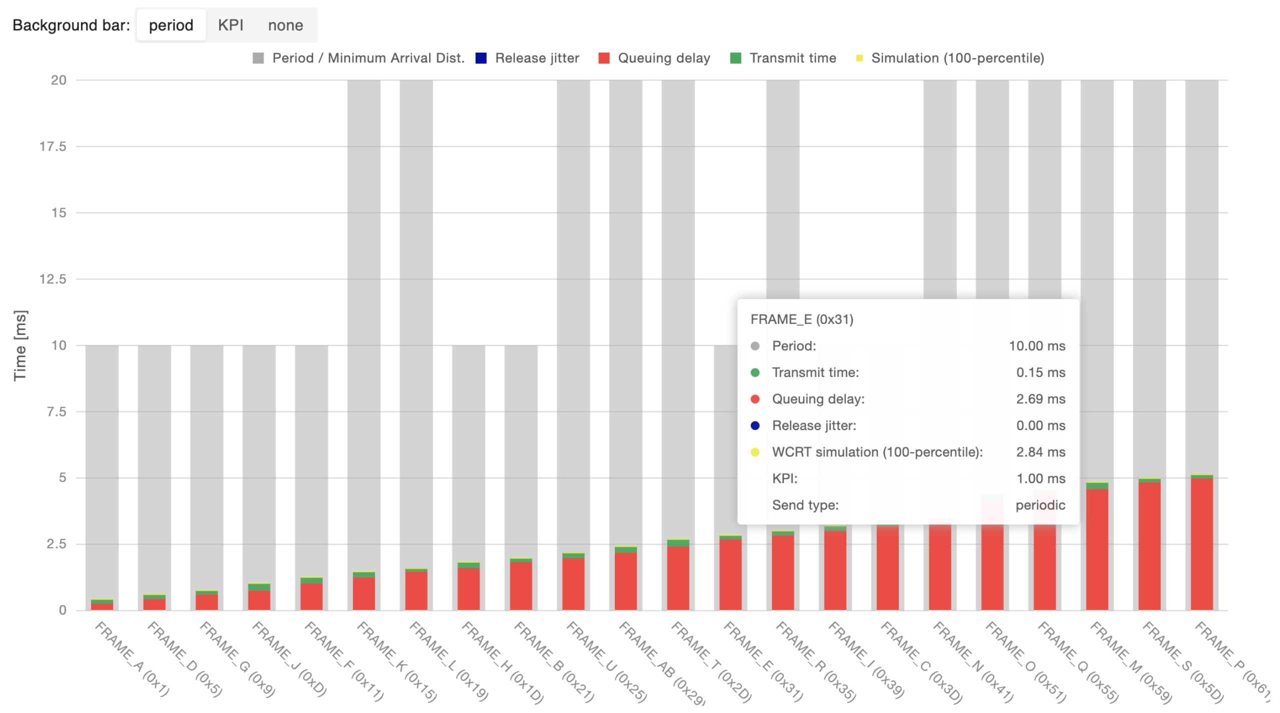
FRAME_AC (0xB5)
- period: 250 ms
- analytical: 9.33 ms
- simulation: 8.50 ms
- status: Valid

This chart allows to validate the reliability of the network in terms of stability of latency for each message. In the chart, every point corresponds to a message. On the x-axis there are increasing message periods (or minimum transmission intervals for non-periodic messages), while the y-axis provides the WCRT time in ms. The set boundary for the maximum acceptable WCRT is represented by a line that separates messages within such a boundary, represented as green points, from messages violating it, represented as red points. In order to better accommodate a wide range of WCRT values, it is possible to choose the type of visualization scale (linear or logarithmic). In the example above, information is reported in logarithmic scale, and the max WCRT is set to be 25% of the period of the messages.

The WCRT is available both according to the analytical model and using the statistical simulation's results, toggling the "WCRT bar" selector.

Hovering on a specific message it is possible to get its details: its period, its analytical WCRT and the simulated WCRT at the specified percentile.

Analytical WCRT breakdown



FRAME_E (0x31)

● Period:	10.00 ms
● Transmit time:	0.15 ms
● Queuing delay:	2.69 ms
● Release jitter:	0.00 ms
● WCRT simulation (100-percentile):	2.84 ms
KPI:	1.00 ms
Send type:	periodic

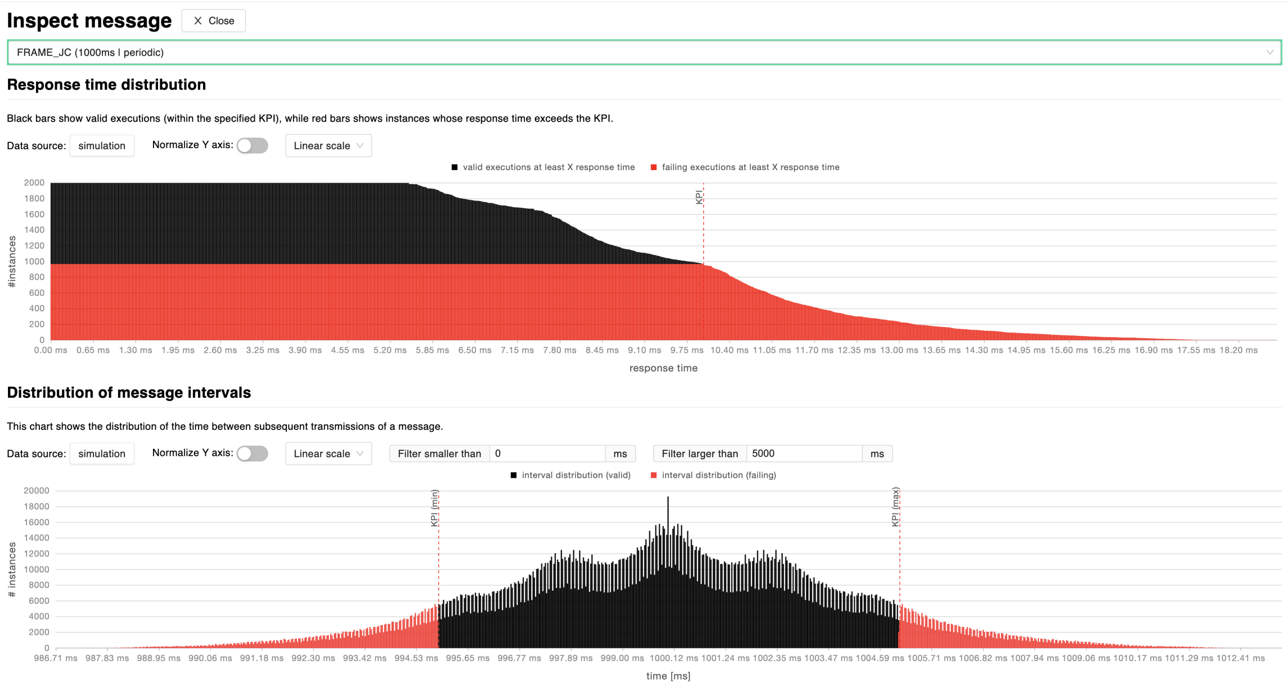
To further inspect the composition of the WCRT summarized in the previous charts, it is possible to analyze the breakdown of the analytical WCRT for each message into its components possibly comparing the result either with the message period/minimum transmission interval or the set KPI boundary on its WCRT.

Clicking on a bar will open the relative message inspection.

Message details

All the information summarized in the previous charts can be inspected in detail for each message by searching the message in the table present in this section. Clicking on the magnifying glass icon will open the relative message inspection.

Message inspection



Inspect message X Close

FRAME_JC (1000ms I periodic)

Response time distribution

Black bars show valid executions (within the specified KPI), while red bars shows instances whose response time exceeds the KPI.

Data source: simulation Normalize Y axis: Linear scale

Distribution of message intervals

This chart shows the distribution of the time between subsequent transmissions of a message.

Data source: simulation Normalize Y axis: Linear scale Filter smaller than 0 ms Filter larger than 5000 ms

valid executions at least X response time
failing executions at least X response time
KPI
#instances
response time
interval distribution (valid)
interval distribution (failing)
KPI (min)
KPI (max)
# instances
time [ms]

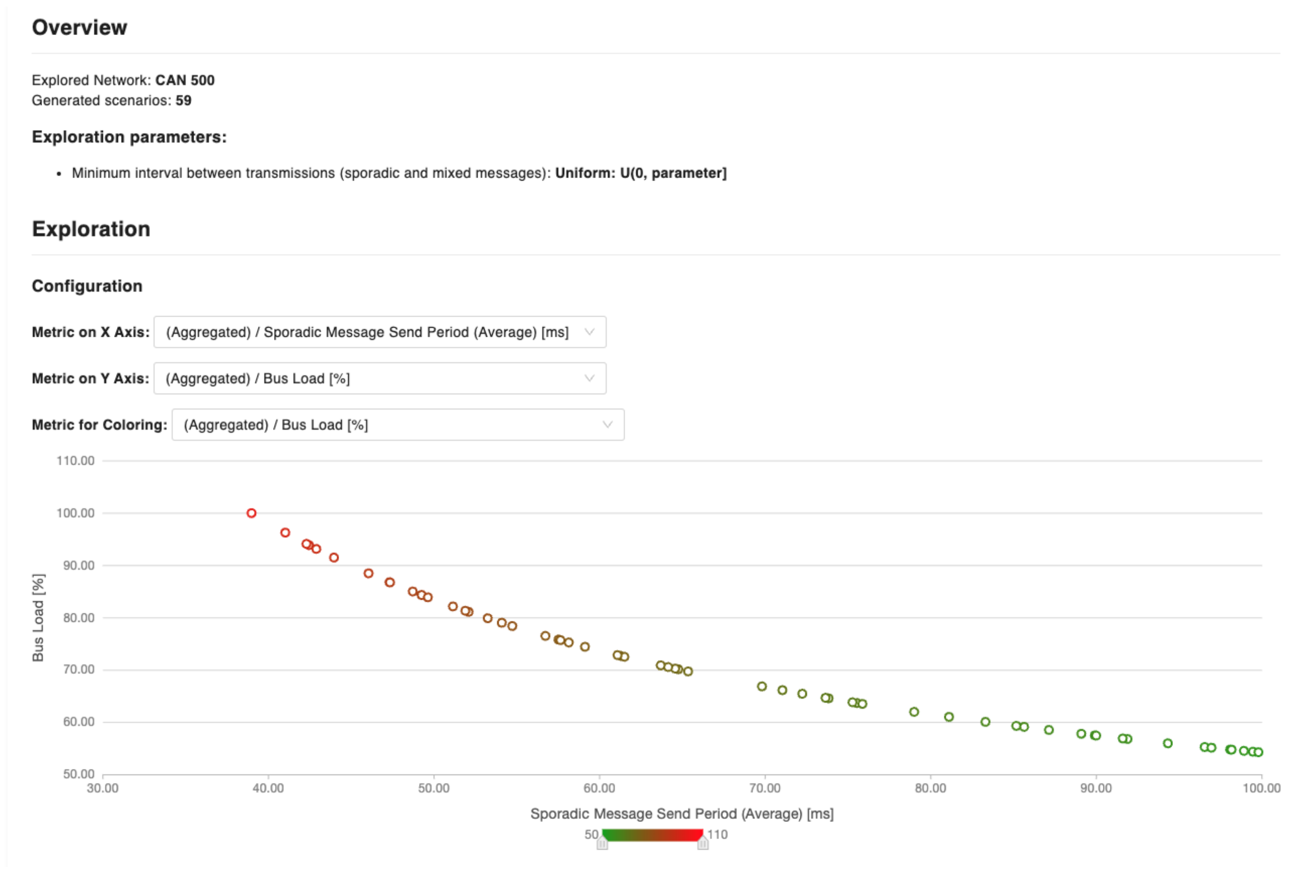
By clicking on a data-point of the charts described in previous sections, it is possible to inspect simulation results for the selected message. The simulation tests a high number of different executions to evaluate the probability of failing user-defined KPIs. This pane's charts help the user navigate through the resulting probability distributions in order to understand the level of reliability for that specific message's transmission timing.

- **Response time distribution**: this chart shows the number of valid (black) and invalid (red) execution instances for which the message falls under a certain response time, for each response time. A dashed red line marks the maximum acceptable response time based on user-defined KPIs. It is possible to normalize the y-axis by switching to a percentage scale and switch between linear and logarithmic scales.

- **Distribution of time intervals**: this chart shows the distribution of the time between subsequent transmissions of a message, highlighting the parts of distribution which are failing (red) user-defined KPIs and those who are valid (black). It is possible to normalize the y-axis by switching to a percentage scale and switch between linear and logarithmic scales. In addition to that, the user can also set upper and lower bound time filters.

Exploration

This section allows to stress test your network to understand possible scenarios that could lead to unintended network occupancies. Provided a scenario in which network and messages have been defined, define a global jitter and a probability distribution to launch a network performance exploration. The analysis will generate multiple scenarios following a probability distribution that will modify the minimum interval between transmissions of sporadic and mixed messages, also check the impact over the network. The report is pretty useful to see, for example, the bus load of network where there are messages that can be originated by a driver/user (i.e. non-periodic).



Overview

Explored Network: **CAN 500**
Generated scenarios: **59**

Exploration parameters:

- Minimum interval between transmissions (sporadic and mixed messages): **Uniform: U(0, parameter]**

Exploration

Configuration

Metric on X Axis: (Aggregated) / Sporadic Message Send Period (Average) [ms]

Metric on Y Axis: (Aggregated) / Bus Load [%]

Metric for Coloring: (Aggregated) / Bus Load [%]

The generated chart is configurable as you have the opportunity to customize the information by modifying the x and y-axis. In the example above, the chart has been configured with the average sporadic/mixed message send period in ms on the x-axis, while the y-axis shows the % bus load. Furthermore, a metric for coloring has been indicated to show how the bus load increases with the increase of the delivery frequency of messages. Each point of the chart, corresponds to a scenario, where messages have been sent with the average period defined in the x-axis. For each scenario (data point), it is possible to obtain detailed information at the message granularity (by obtaining the Response Time Analysis and Relative response time violations charts)

Huxelerate Network Inspector

The **Huxelerate Network Inspector** is a utility that allows to inspect a given network trace file (BLF) and the corresponding DBC to produce a **.hnet** file that collects metrics and statistics on messages behavior.

To generate an **.hnet**, please download the hux-network-inspector.

To get comprehensive information on all the arguments to pass to the utility tool, run the following:

Windows:

```
hux-network-inspector.exe --help
```

Usage Example

A common usage example of the tool can be found below:

```
hux-network-inspector.exe \
  -d network.dbc \
  -b trace.blf \
  -c 0 \
  -a 500000 \
  -o results.hnet \
  -m 10000
```

HNET upload

Once the **.hnet** file generation is complete, it can be uploaded to the platform to visualize all statistical data of the network messages.

When opening a network analysis, click on the **Upload HNET** button and choose the HNET file from your system.

As soon as the upload finishes, you may visualize the traced data on the graphs alongside with the network analysis data.

There also is a detailed table containing the messages statistics, both for analytical, simulated and traced data.

Note

If the DBC file used to populate the network frames is not the same used to generate the HNET file, there may be mismatches, or some of the traced data may be missing.

Huxelerate Network Unpacker

The **Huxelerate Network Unpacker** is a utility that allows to unpack CAN messages incapsulated in an Ethernet frame, following Autosar Bus Mirroring specification . Starting from a BLF file, it generates a new BLF containing the unpacked CAN messages, and optionally all the messages in the input BLF.

To unpack messages from a **.blf** file, please download the hux-network-unpacker.

To get comprehensive information on all the arguments to pass to the utility tool, run the following:

Windows:

```
hux-network-unpacker.exe --help
```

Usage Example

A common usage example of the tool can be found below:

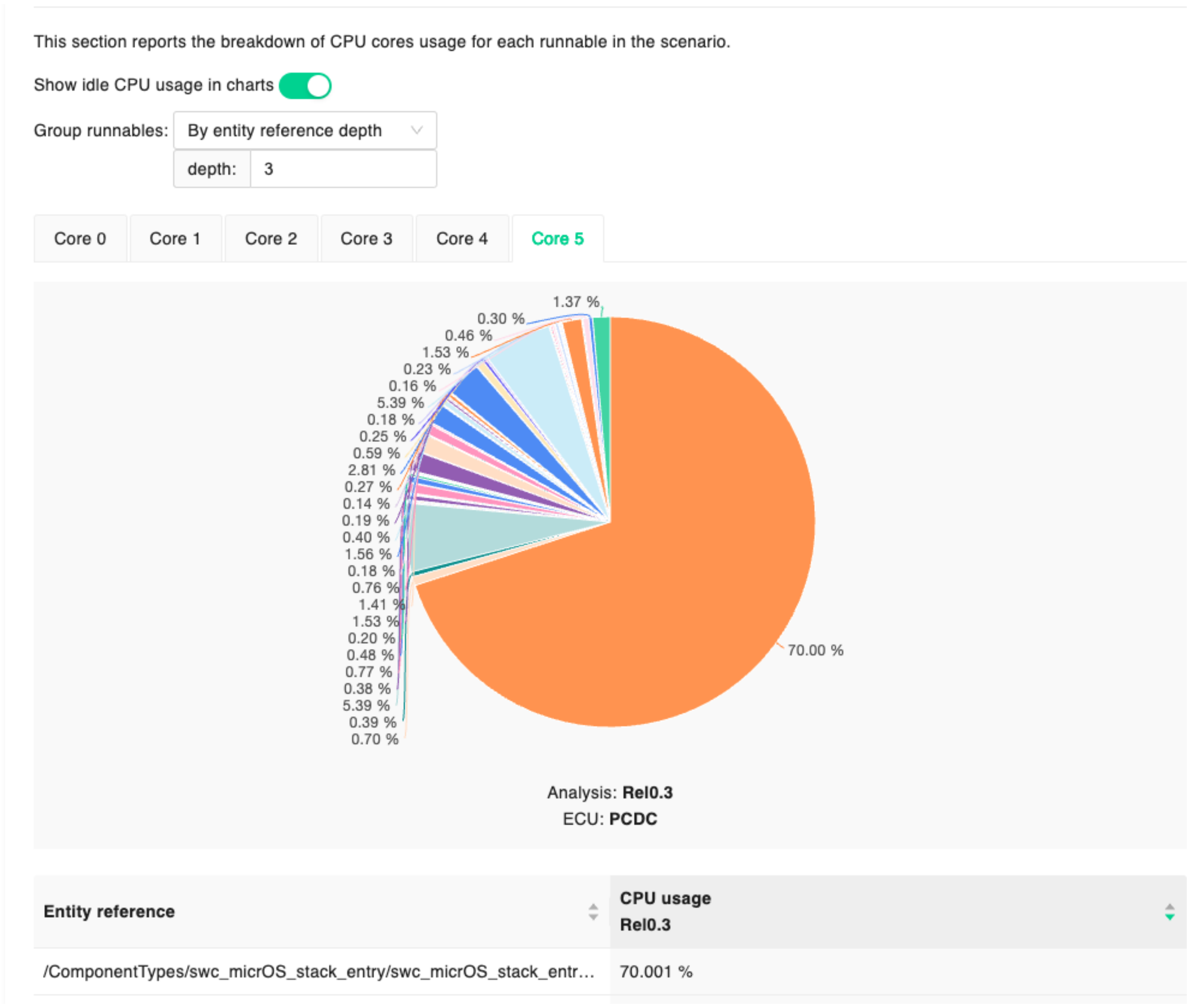
```
hux-network-unpacker.exe \
  -b input_with_packed_messages.blf \
  -o output_with_unpacked_messages.blf
```

Note

In case of issues with license validation, the **Huxelerate Network Unpacker** offers a flag to skip SSL verification with the –*skip-verify-ssl* option.

Resource Analysis

Resource analyses provide information about the resource utilization for a specific scenario.



This section reports the breakdown of CPU cores usage for each runnable in the scenario.

Show idle CPU usage in charts

Group runnables: By entity reference depth

depth: 3

Analysis: **Rel0.3**
ECU: **PCDC**

Entity reference	**CPU usage Rel0.3**
/ComponentTypes/swc_micrOS_stack_entry/swc_micrOS_stack_entr...	70.001 %

The report is mainly composed of two sections:

- an overview of the resource utilization
- a breakdown of the CPU usage

The overview provides current and achievable CPU usage per each core of the analyzed ECU Type, while the breakdown provides more detailed information on resource occupancy of each periodic runnable. In the second section of the report, it is possible to group runnables to get resource usage information of groups of software (e.g. Autosar compositions).

Timing Analysis

A Timing analysis is the result of the simulation of a specific scenario defined in the system definition section. The simulation involves all the information provided in the scenario (ECUs, networks, software components, task mapping and timing constraints), and allows to analyze the system in its entirety, providing information on:

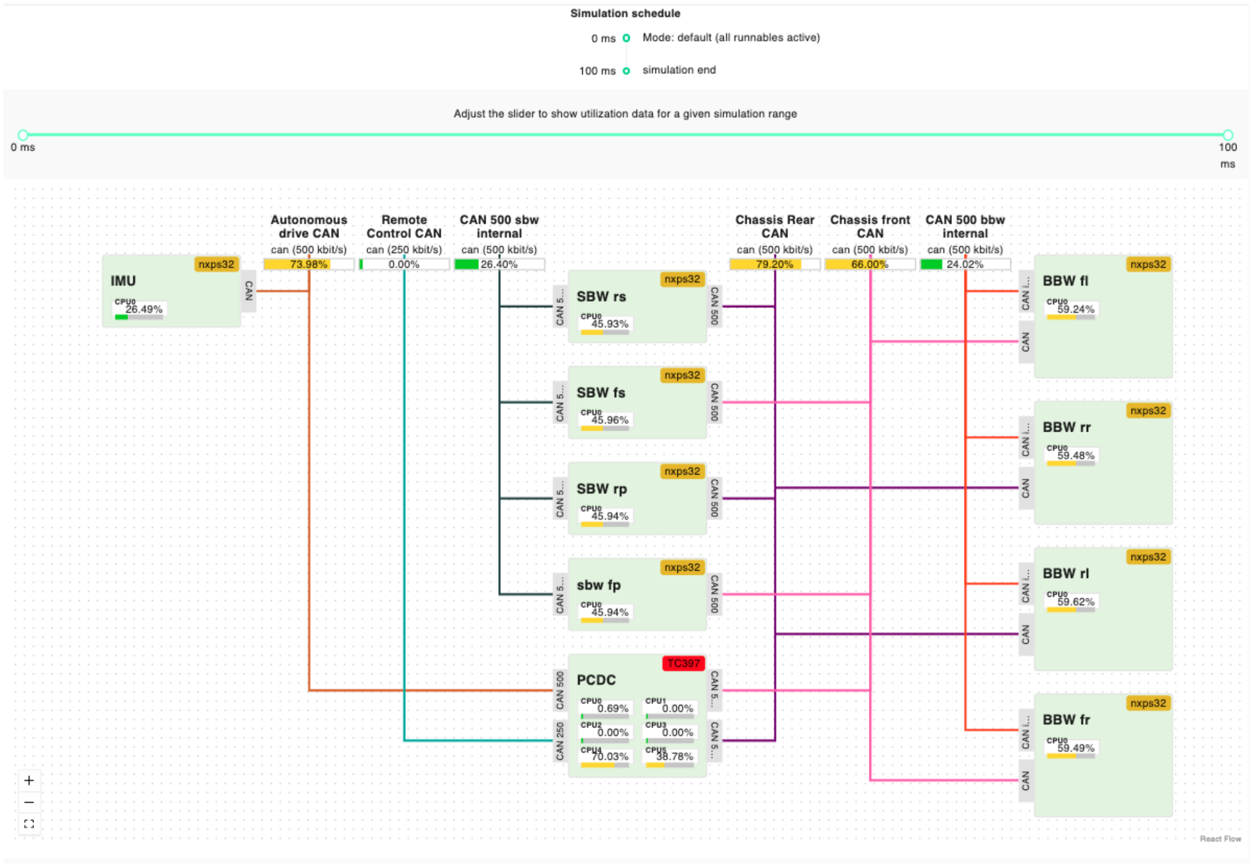
- resources occupancy
- timing constraints violation
- interactions among software functions (e.g. preemption)
- information about software functions (e.g %time executing, %time ready)

Timing analyses can be launched on different scenarios, and different task mappings. Software function execution data is taken from the runnable performance reports originated from the HPROFs, and their periodicity can be simulated with a Gaussian distribution, or by using always the longest execution time. It is possible to define a **simulation schedule** (i.e. which runnable/function is active and when). By default, all runnables/functions are active during the simulation however, to specify a different behavior where specific runnables/functions are active, it is possible to define states in the operation modes section, and specify the transition upon analysis launch.

The last piece of information to provide for performing a timing analysis is the profiling data, that is the specific version of the performance report to use for the analysis.

Timing Analysis report

The report presents a hierarchical view of the performance information of the simulation. The first level of information is represented by a graphical view of the occupancy of each resource (network, ECU).

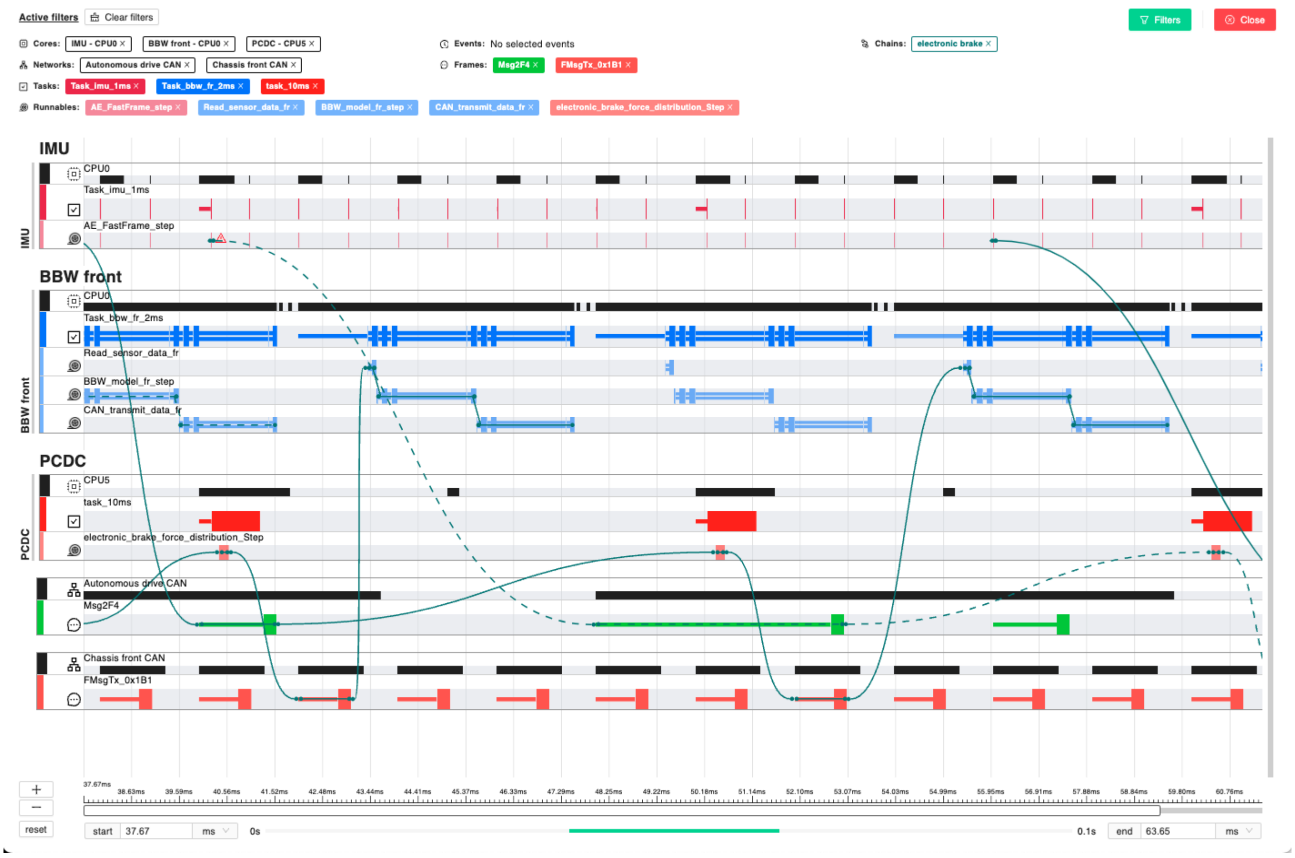

Simulation schedule
0 ms Mode: default (all runnables active)
100 ms simulation end
Adjust the slider to show utilization data for a given simulation range
0 ms
100 ms
Autonomous drive CAN can (500 kbit/s) 73.98%
Remote Control CAN can (250 kbit/s) 0.00%
CAN 500 sbw internal can (500 kbit/s) 26.40%
Chassis Rear CAN can (500 kbit/s) 79.20%
Chassis front CAN can (500 kbit/s) 66.00%
CAN 500 bbw internal can (500 kbit/s) 24.02%
IMU nxps32 CPU0 26.49%
SBW rs nxps32 CPU0 45.93%
SBW fs nxps32 CPU0 45.96%
SBW rp nxps32 CPU0 45.94%
sbw fp nxps32 CPU0 45.94%
PCDC TC397 CPU0 0.69% CPU1 0.00% CPU2 0.00% CPU3 0.00% CPU4 70.03% CPU5 38.78%
BBW fl nxps32 CPU0 59.24%
BBW rr nxps32 CPU0 59.48%
BBW rl nxps32 CPU0 59.62%
BBW fr nxps32 CPU0 59.49%
CAN
CAN 5...
CAN 500
CAN 250
CAN i...
React Flow

The chart shows the interconnections, and percentage occupancy of each core of the ECUs, and networks involved in the simulation. Each occupancy is flagged with a specific color, to highlight potential problems linked with resource occupancy.

Occupancy information is detailed in a tabular format both for compute and network resources.Compute resources are organized for each ECU and corresponding core, showing information for each task, regarding the percentage of time in the running, ready and suspended states. By expanding the specific task, it is possible to see the detailed information about a specific function.

Regarding network resources the information provided indicates the percentage of running and ready time for each message, on the specific network resource.

The final part of the report regards event chains. Within the simulation, the system checks whether, the timing constraints are met for each event chain, providing its success rate. By clicking on the specific event chain, it is possible to observe a gantt chart with the corresponding interactions, and (if present) the specific failing instances.


Active filters | Clear filters

Cores: IMU - CPU0 × | BBW front - CPU0 × | PCDC - CPU5 ×

Networks: Autonomous drive CAN × | Chassis front CAN ×

Tasks: Task_imu_1ms × | Task_bbw_fr_2ms × | task_10ms ×

Runnables: AE_FastFrame_step × | Read_sensor_data_fr × | BBW_model_fr_step × | CAN_transmit_data_fr × | electronic_brake_force_distribution_Step ×

Events: No selected events

Frames: Msg2F4 × | FMsgTx_0x1B1 ×

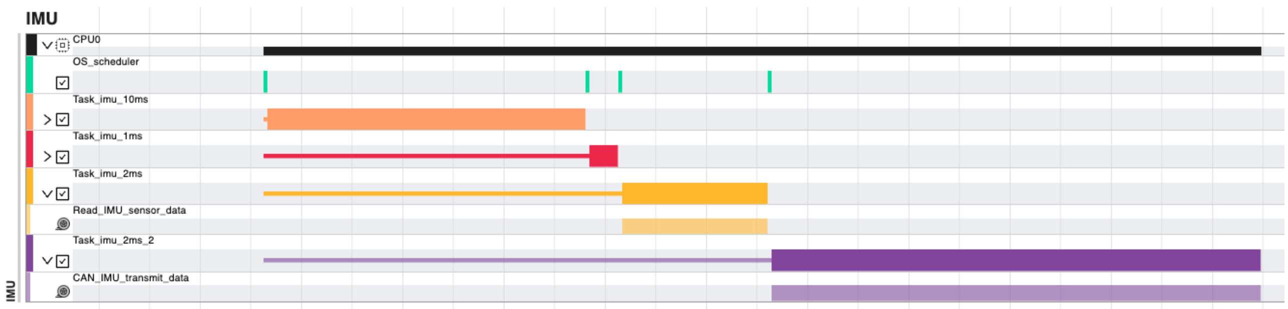
Chains: electronic brake ×

start 37.67 ms | 0s | 0.1s | end 63.65 ms

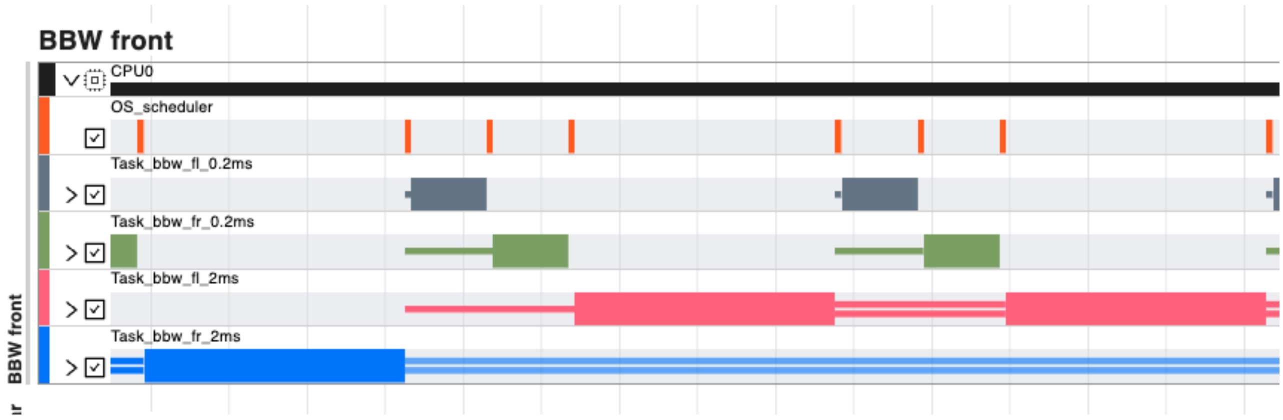
Failing event chains are shown with a dotted line, and, by clicking on the line, it is possible to get more details about it. The gantt chart shown by clicking on the action button near an event chain, abstracts all the information about the simulation to focus on the specific timing requirement. However, it is possible to get the whole simulation data by clicking the “Show Timing Gantt” button.

The “Show Timing Gantt” button provides the complete simulation data in a hierarchical way. The top level is represented by the ECUs and the network, and it is possible to deep dive into the runnable/function execution. To focus the visualization, it is possible to use the filter at the top-right of the screen. The status of a task or runnable/function is represented by three possible views, as explained in the examples below:

- rectangles indicate a running task/function/runnable
- single lines indicate a task/function/runnable that is ready, waiting for some other element with higher priority
- double lines indicate a task/function/runnable that is suspended by some other element with higher priority



In the picture above, Task_imu_2ms stays in ready state as Task_imu_10ms and Task_imu_1ms have higher priorities, and then transition ready when the aforementioned tasks terminates execution. This interaction is shown in the gantt with a single yellow line, followed by the yellow rectangle.



In this second example, it is observable how Task_bbw_fl_2ms gets preempted by the Tasks ad 0.2ms. In this case the interaction is shown with double lines.

Integrating with a CI/CD/CT System

Huxelerate SDV Performance can be seamlessly integrated into Continuous Integration (CI), Continuous Delivery (CD), and Continuous Testing (CT) systems. This integration enables automatic performance analysis of your software and hardware architecture whenever changes are made to the codebase.

The following sections provide guidelines and examples for building a CI/CD/CT pipeline that incorporates Huxelerate SDV Performance.

Common Pipeline Prerequisites

To create the pipelines described in this document, ensure the following:

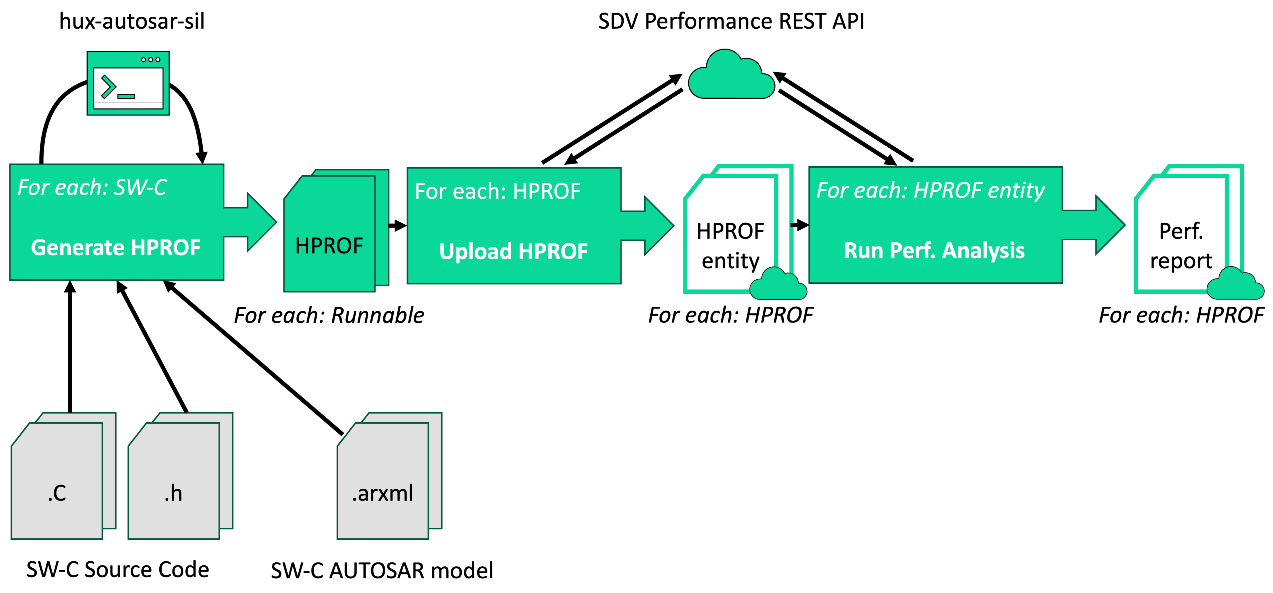
- An Huxelerate SDV Performance platform Normal user account to execute the pipeline. Refer to the Organization Settings section for instructions on creating a user.
    - The pipeline will use this user's **REST API token** as a **secret** for platform authentication. Refer to the Settings section to learn how to generate a REST API token.
- An Huxelerate SDV Performance platform Project to store performance reports. Refer to the System Definition section for project creation details.
    - The pipeline will use this project's **ID** as a **parameter** to upload HPROF files and retrieve performance reports.
    - The project must include at least one ECU Type for performance analysis.

Interactive REST API Documentation

Explore all REST API endpoints interactively at: Huxelerate SDV Performance API Documentation.

Pipeline: Performance Analysis of AUTOSAR-Compliant Software

This pipeline targets AUTOSAR-compliant software and generates performance reports for each Runnable Entity in the software.



Pipeline Additional Prerequisites

To implement this pipeline, you will need:

- The latest version of the Huxelerate autosar utility installed on the system running the pipeline.
    - You can download it from the Resources section.
    - The Huxelerate autosar utility generates HPROF files and uploads them to the platform.
    - (Optional) If using the cloud for HPROF generation, ensure the system has access to the required cloud storage buckets (see HPROF Generation as-a-service).

Additionally, for each Software Component Type to analyze, you must know:

- The location of the software component's source code (implementation and header files) in your codebase.
- The location of the ARXML description models of the software component in your codebase.
- (Optional) A CSV file with input data for analysis. Refer to the generate_csv_template command documentation for details.

Step 1: Generate HPROF Files for a Software Component Type

Execution Scope: One execution per Software Component Type in your codebase.

Inputs:

- Source code of the software component (implementation and header files).
- ARXML description models of the software component.
- (Optional) CSV file with input data for analysis.

Outputs:

- A LOG file detailing the generation process (success or failure).
- For each Runnable Entity in the Software Component Type, one HPROF file (if successful).

Use the generate_hprof command of the Huxelerate autosar utility to generate HPROF files.

Hint: If using the cloud for HPROF generation, refer to the relevant documentation for CLI usage instructions.

Step 2: Upload HPROF Files to the Platform

Execution Scope: One execution per HPROF file generated in Step 1.

Inputs:

- REST API token of the user executing the pipeline.
- Project ID where the HPROF file will be uploaded.
- The HPROF file to upload.

Outputs:

- Details of the uploaded HPROF file (ID, name, etc.).

Upload HPROF files using the REST API endpoint:

```
PUT http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
hprofs
```

You can associate tags with each HPROF file for easier searching and filtering. Useful tags include:

- Software Composition name(s) associated with the HPROF file.
- Version of the Software Component Type or System associated with the HPROF file.

Step 3: Run Performance Analysis

Execution Scope: One execution per HPROF file uploaded in Step 2.

Inputs:

- REST API token of the user executing the pipeline.
- Project ID where the HPROF file was uploaded.
- HPROF file ID (from Step 2).

Outputs:

- Details of the analysis (ID, name, etc.).

Run performance analysis using the REST API endpoint:

```
PUT http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
runnable-analyses/
```

Specify a target hardware (e.g., an ECU Type core) for the analysis. Retrieve available ECU Types and cores using the REST API endpoint:

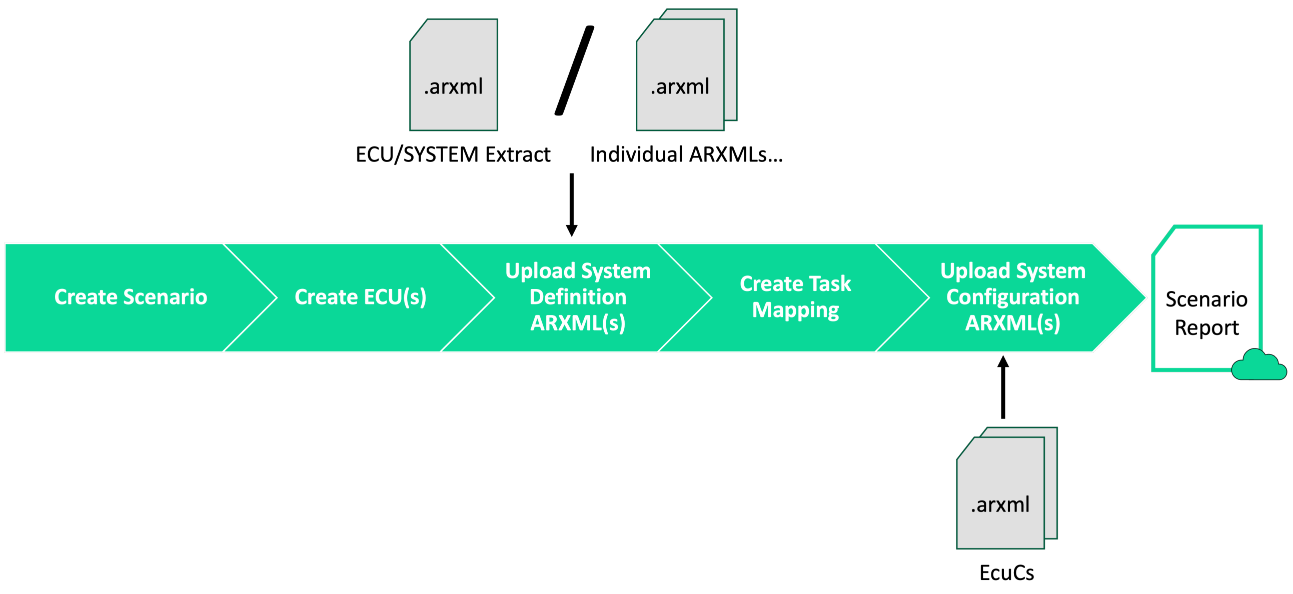
```
GET http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
ecu-types/
```

Result

After completing this pipeline, performance analysis results will be available in the specified project.

Pipeline: Architectural Analysis of AUTOSAR-Compliant Systems

This pipeline targets AUTOSAR-compliant systems and provides structural details independently of the software implementation. Unlike the previous pipeline, it does not require HPROF file generation but relies on ARXML description models.



Pipeline Additional Prerequisites

To implement this pipeline, you will need:

- ARXML description models of the system:
    - Either a single ARXML *ECU Extract* or *SYSTEM Extract* file.
    - Or multiple ARXML files collectively describing the system (e.g., files used in the previous pipeline).
- ECU Configuration ARXML files describing, for each ECU in the system, the Runnable Entities (grouped into Tasks) it executes. These files are typically generated by your organization's ECU Configuration tool (e.g., Vector DaVinci).

Step 1: Create a New Scenario

Create a new scenario to reflect the system's structure. Use the REST API endpoint:

```
POST http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios
```

Scenarios can be created empty or by copying an existing scenario's structure. In the latter case, some subsequent steps may be unnecessary.

Step 2: Create ECU(s)

Execution Scope: One execution per ECU in the system.

Create one ECU entry for each ECU in your system using the REST API endpoint:

```
POST http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/ecus
```

Specify the ECU Type when creating an ECU. Retrieve available ECU Types using the REST API endpoint:

```
GET http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
ecu-types/
```

Hint: If the scenario was created by copying an existing scenario, this step may be unnecessary.

Step 3: Upload the System Definition

Upload ARXML description models of the system using the REST API endpoint:

```
POST http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/system-definition/jobs
```

This endpoint accepts:

- A single ARXML file (*ECU Extract* or *SYSTEM Extract*).
- Multiple ARXML files in a Zip archive.

Specify the following parameters:

- `system_definition_entity_type = "swc"` to indicate a Software Component-based System Definition.
- `description_standard = "ARXML"` for ARXML format files.

Optional parameters to resolve conflicts during upload:

- `update_duplicates`: Update existing elements in the scenario.

- `regenerate_unsupported_uuid`: Regenerate non-UUID4-compliant UUIDs.
- `regenerate_duplicated_uuid`: Regenerate duplicated UUIDs.

Check upload status using the REST API endpoint:

```
GET http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/system-definition/jobs/{id_job}
```

Confirm successful uploads using:

```
PATCH http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/system-definition/jobs/{id_job}

Body: {"confirm": true}
```

Clear failed jobs using:

```
DELETE http://api.sdvperformance.huxelerate.it/v1.0/projects/
{id_project}/scenarios/{id_scenario}/system-definition/jobs/{id_job}
```

Step 4: Create a New Task Mapping

Create a Task mapping to map Runnable Entities to ECUs and their cores. Use the REST API endpoint:

```
POST http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/mappings
```

Task mappings can be created empty or by copying an existing mapping's structure. In the latter case, some subsequent steps may be unnecessary.

Step 5: Upload Task Mapping from ECU Configuration ARXML Files

Upload ECU Configuration ARXML files to populate the Task mapping using the REST API endpoint:

```
POST http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/mappings/{id_mapping}/task-mapping/jobs
```

This endpoint accepts:

- A single ARXML file containing the ECU Configuration.
- Multiple ARXML files in a Zip archive.

Specify the following parameters:

- `system_definition_entity_type = "ecuc"` to indicate an ECU Configuration-based Task mapping.
- `description_standard = "ARXML"` for ARXML format files.
- `id_mapping`: Target Task mapping ID.
- `id_ecu`: Target ECU ID (repeat for each ECU in the system).

Optional parameters to resolve conflicts during upload:

- `regenerate_unsupported_uuid`: Regenerate non-UUID4-compliant UUIDs.
- `regenerate_duplicated_uuid`: Regenerate duplicated UUIDs.

Check upload status using the REST API endpoint:

```
GET http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/mappings/{id_mapping}/task-mapping/jobs/{id_job}
```

Confirm successful uploads using:

```
PATCH http://api.sdvperformance.huxelerate.it/v1.0/projects/{id_project}/
scenarios/{id_scenario}/mappings/{id_mapping}/task-mapping/jobs/{id_job}

Body: {"confirm": true}
```

Clear failed jobs using:

```
DELETE http://api.sdvperformance.huxelerate.it/v1.0/projects/
{id_project}/scenarios/{id_scenario}/mappings/{id_mapping}/task-mapping/
jobs/{id_job}
```

Result

After completing this pipeline, explore the interactive System Structure report at:

```
https://sdvperformance.huxelerate.it/projects/{id_project}/scenarios/
{id_scenario}/system/scenario
```

Replace `{id_project}` and `{id_scenario}` with the respective IDs used in the pipeline.

Platform and tools licensing

Huxelerate SDV performance plarform requires a license to be used. The license is necessary to access the web platform, and to use the on-premise tools. To require a license, please contact your representative at Huxelerate, or write to info@huxelerate.it.

License for on-premise tools

On-premise tools requires a license to be used on a local workstation, or on a server (i.e. CI/CD). The tool performs a license check that requires communication with the Huxelerate license server, and is directly linked with a specific user account.

License validation can be performed in two ways: 1. via **token** 2. via **license file**

Token-based license validation

Token-based license validation requires the user to:

- generate a token from the Huxelerate web platform (ref: Settings)
- provide the token in command line when running the tool or using the HUXELERATE_LICENSE_TOKEN environment variable

here's an example of running the Network Unpacker tool with a token:

```
hux-network-unpacker.exe -b input.blf --license-token <your-token>
```

The flags for license token and file are consistant across different tools. Here's another example for Huxelerate Autosar utility:

```
hux-autosar-sil.exe --license-token <your-token> [other options...]
```

License file-based validation

License file-based validation requires the user to:

- download the license file from the Huxelerate web platform (ref: Settings)
- place the license file in the ~/.huxelerate directory (Linux) or in the %USERPROFILE%\.huxelerate directory (Windows)

- run the tool without providing any license information

Alternatively, it is possible to place the license file in a custom directory and provide the path to the directory in command line when running the tool, or using the HUXELERATE_LICENSE_PATH environment variable.

Resources

Resources can be downloaded from the platform