



MBSE Implement

ASN.1 and AADL Generators Software User Manual

MBSE-N7S-UM-19003 rev. 1.5

N7 SPACE SP. Z O.O.

| Prepared by | Date and Signature |
|-------------------|--------------------|
| Michał Kurowski | |
| Verified by | |
| Konrad Grochowski | |
| Approved by | |
| Michał Mosdorf | |

Table of contents

| | |
|--|----|
| 1. Introduction | 5 |
| 1.1. Scope | 5 |
| 1.2. Project objectives | 5 |
| 2. Applicable and reference documents..... | 7 |
| 2.1. Applicable documents | 7 |
| 2.2. Applicable standards | 7 |
| 2.3. Reference documents | 7 |
| 3. Terms, definitions and abbreviated terms | 8 |
| 4. Installation | 9 |
| 4.1. General | 9 |
| 4.2. Prerequisites | 9 |
| 4.3. Installation procedure | 10 |
| 5. Configuration..... | 11 |
| 6. Usage | 14 |
| 6.1. Data Model | 14 |
| 6.2. Architecture | 17 |
| 6.3. Additional TASTE integration tools | 22 |
| 6.4. Example workflow | 22 |
| 6.4.1. Define the data model..... | 22 |
| 6.4.2. Define the architecture | 23 |
| 6.4.3. Apply the required properties | 24 |
| 6.4.4. Export the data model..... | 25 |
| 6.4.5. Export the architecture | 26 |
| 6.4.6. Postprocessing, editing and compilation | 27 |
| 7. Supported Capella Features | 30 |
| 7.1. Data Model | 30 |
| 7.2. System Model..... | 31 |
| 8. Best practices..... | 32 |
| 8.1. Naming convention | 32 |
| 8.2. Inheritance | 32 |
| 8.3. Cardinality | 32 |
| 8.4. Characters and strings | 32 |
| 8.5. Nesting | 32 |
| 8.6. Communication layers..... | 32 |



| | |
|--|----|
| 8.7. GUI considerations..... | 33 |
| 8.8. TASTE knowledge and awareness of implementation details | 33 |
| 9. Messages | 34 |
| 10. Lists | 37 |
| 10.1. List of tables | 37 |
| 10.2. List of figures | 37 |



Change Record

| Issue | Date | Changes |
|-------|------------|--|
| 1.0 | 2019-06-19 | Initial release |
| 1.1 | 2019-09-12 | Updated information relevant to Expression handling Updated information relevant to AADL generation Updated general installation information |
| 1.2 | 2019-09-18 | Updated plugin version Updated error messages |
| 1.3 | 2019-09-20 | Added best practices relevant to AADL generation Added Capella features supported by AADL generator |
| 1.4 | 2020-01-31 | Updated with information relevant to plugin version 1.3.4 compatible with Kazoo |
| 1.5 | 2020-02-18 | Added information regarding GUI merging Updated information regarding Capella version compatibility |

1. Introduction

1.1. Scope

This document constitutes the software user manual for the following software deliverables from “MBSE Implement” project:

- ASN.1 Generator;
- AADL Generator.

The reader should be familiar with:

- Capella and Arcadia method [RD1];
- TASTE and ASN.1 [RD2].

The following introduction provides a short description of the project objectives.

1.2. Project objectives

One of the major objectives of the project is the development of ASN.1 and AADL Generators capable of converting data and architecture models created in Capella into models compatible with TASTE.

Capella [RD1] is an open-source modelling environment for Model Based Systems Engineering (MBSE), originally created in 2007 by Thales and then released in 2015 as an Eclipse project by PolarSys. It supports the Arcadia modelling method, also designed by Thales. The method guides the user through the following “working levels”, each of which is supported by Capella:

- Operational Analysis;
- Analysis of the System Needs;
- Logical Architecture;
- Physical Architecture;
- End Product Breakdown Structure and Integration Contracts.

At each working level, the model can be visualized and manipulated through a variety of diagrams, including:

- breakdown diagrams;
- interaction diagrams;
- scenario diagrams;
- role diagrams;
- architecture diagrams;
- class diagrams,
- mode and state machine diagrams.

The important feature of Capella is that although different diagrams provide different views, within each project there is a single, coherent underlying model.

TASTE [RD2] is a model-driven tool-chain targeting heterogeneous, embedded systems. It was created in 2008 under the initiative of the European Space Agency and consists of various tools facilitating creation of systems using formal models and automatic code generation. The key technologies involved are AADL for architecture definition, ASN.1 for data modelling and SDL for behaviour specification. Supplementary languages and technologies include, but are not limited to:

- Simulink;
- SCADE;
- Ada;
- C/C++;
- Python;
- CHEDDAR;
- MAST;
- MSC.

The basic TASTE workflow includes:

- definition of data models using ASN.1;
- definition of a logical architecture using an Interface View description in AADL;
- definition of a physical architecture using a Deployment View description in AADL;
- definition of behaviour using various technologies, such as SDL.

ASN.1 and AADL Generators developed within the scope of this project aim to translate the relevant part of the Capella model (which can be expressed via class diagrams and various architecture diagrams) into data and architecture models understood by TASTE. It must be noted that the ASN.1 dialect supported by TASTE is a subset of the ASN.1 standard. Additionally, in order to properly interact with the user via a GUI, TASTE introduces additional properties to the supported AADL dialect.

2. Applicable and reference documents

2.1. Applicable documents

| ID | Title | Reference | Rev. |
|----|-------|-----------|------|
|----|-------|-----------|------|

2.2. Applicable standards

| ID | Title | Reference | Rev. |
|----|-------|-----------|------|
|----|-------|-----------|------|

2.3. Reference documents

| ID | Title | Reference | Rev. |
|-----|------------------------|---|------|
| RD1 | Capella Wiki | https://wiki.polarsys.org/Capella | N/A |
| RD2 | TASTE community page | https://taste.tuxfamily.org | N/A |
| RD3 | Capella downloads page | https://www.polarsys.org/capella/download.html | N/A |



3. Terms, definitions and abbreviated terms

This document extra specific acronyms and abbreviations are listed here under.

| | |
|---------|--|
| AADL | Architecture Analysis and Design Language |
| ASN1SCC | ESA's ASN.1 compiler |
| ASN.1 | Abstract Syntax Notation One, dialect supported by ASN1SCC |

4. Installation

4.1. General

Both ASN.1 and AADL Generators are delivered as a single Capella plugin – Capella-TASTE Plugin. Capella is provided for Windows, Linux and Mac operating systems. The technical choices made during the plugin development did not introduce any additional platform dependencies and consequently the plugin should work on the same platforms as Capella. However, according to Capella download page [RD3], both Linux and Mac versions of Capella have not been tested. Consequently the plugin is developed and extensively tested only on Windows platform. However, due to the dependency of TASTE on Linux, some plugin tests have been also performed on Linux platform. This document contains screens from both Windows and Linux Capella versions.

4.2. Prerequisites

Table 1 lists all Capella-TASTE Plugin prerequisites.

Table 1 - Capella-TASTE Plugin prerequisites

| Item | Version | Description |
|---------|---------|---|
| Capella | 1.3.1 | Eclipse based IDE that hosts the plugin |

Default Capella version 1.3.0 installation contains all the required Java dependencies. The required Java bundles are as follows:

- org.eclipse.ui;
- org.eclipse.core.runtime;
- org.polarsys.capella.core.platform.sirius.ui.navigator;
- org.junit;
- org.eclipse.sirius.diagram.ui;
- org.eclipse.gef;
- org.eclipse.gmf.runtime.diagram.ui.

The required Java packages are as follows:

- org.eclipse.core.resources;
- org.eclipse.emf.common;
- org.eclipse.emf.common.util;
- org.eclipse.emf.ecore;
- org.eclipse.emf.ecore.resource;
- org.polarsys.capella.common.data.helpers.modellingcore;
- org.polarsys.capella.common.data.helpers.modellingcore.utils;
- org.polarsys.capella.common.data.modellingcore;
- org.polarsys.capella.common.data.modellingcore.impl;
- org.polarsys.capella.common.data.modellingcore.provider;
- org.polarsys.capella.common.data.modellingcore.util;
- org.polarsys.capella.core.data.capellacore;
- org.polarsys.capella.core.data.capellacore.util;
- org.polarsys.capella.core.data.information;
- org.polarsys.capella.core.data.information.datatype;
- org.polarsys.capella.core.data.information.datatype.util;
- org.polarsys.capella.core.data.information.datavalue;
- org.polarsys.capella.core.data.information.datavalue.util.

4.3. Installation procedure

Capella-TASTE Plugin is delivered as a single Jar file that should be placed inside the “plugins” subdirectory of the Capella installation directory. E.g. if the path to eclipse.exe of the Capella installation is

“C:\Capella\eclipse\eclipse.exe”,

then the plugin should be placed inside

“C:\Capella\eclipse\plugins\”

directory.

It must be ensured that only a single version of the Capella-TASTE plugin is present within the plugins directory. The Jar file is named as follows:

capellatastepugin.\$CAPELLA_MAJOR_VERSION.\$CAPELLA_MINOR_VERSION.\$PLUGIN_VERSION.\$BUILD_ID.jar

\$CAPELLA_MAJOR_VERSION indicates the major version of the target host IDE. Should be “1”.

\$CAPELLA_MINOR_VERSION indicates the minor version of the target host IDE. Should be “3”.

\$PLUGIN_VERSION indicates Capella-TASTE plugin version. This manual refers to version “4”.

\$BUILD_ID indicates build ID.

5. Configuration

Capella-TASTE Plugin adds “TASTE” preferences page (presented in Figure 1), which is accessible at the standard preferences location, under Window->Preferences.

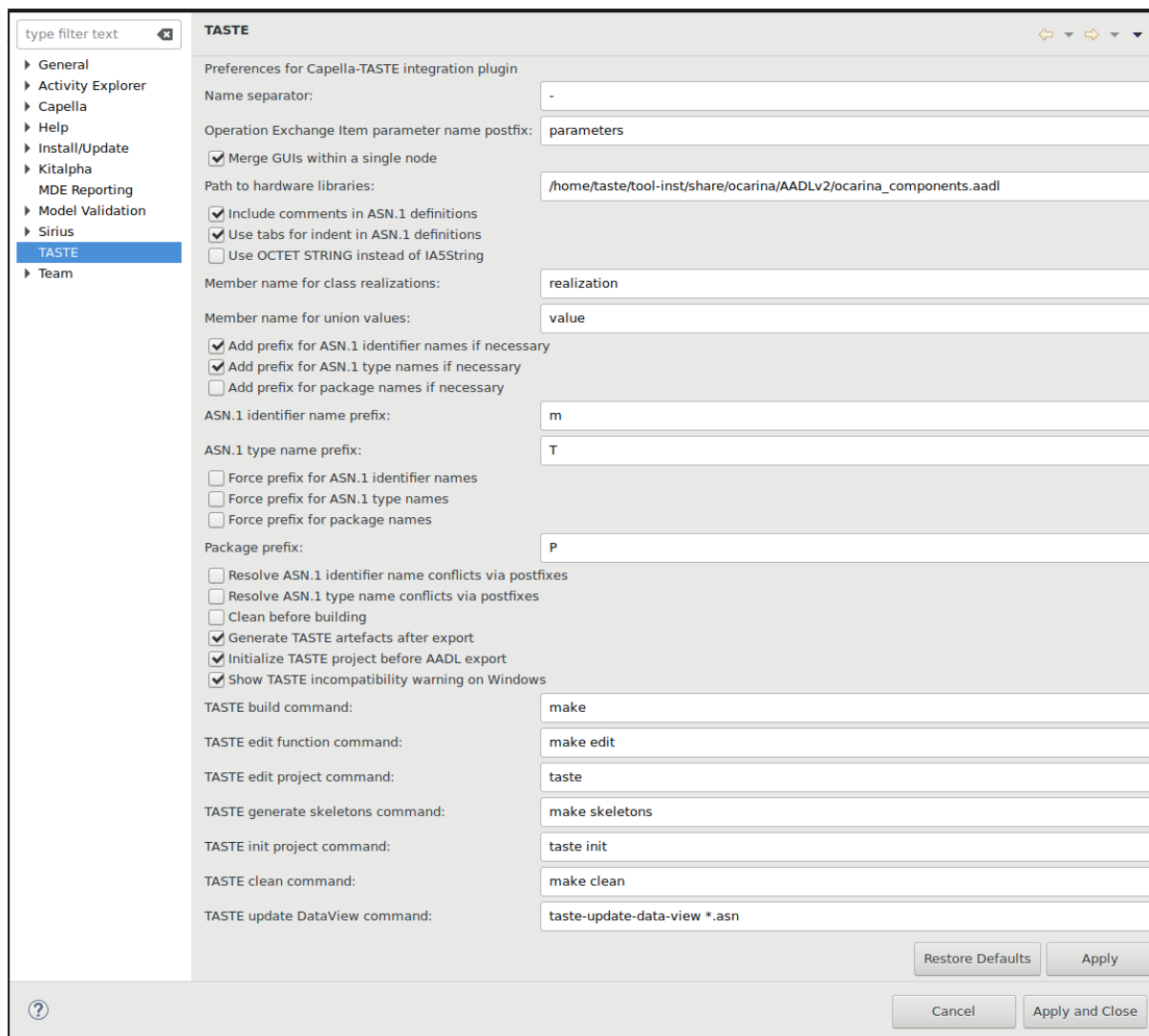


Figure 1 - Plugin preferences page

The preferences page contains the following configuration options:

- *Name separator:*
Separator used for concatenating parts into the resulting names. E.g.:
FirstNamePart-SecondNamePart
- *Operation Exchange Items parameter name postfix:*
A single Exchange Item with exchange mechanism set to Operation may have subitems assigned to different categories, such as IN, OUT, INOUT, RETURN or EXCEPTION. Such Exchange Items are translated into several ASN.1 sequences, one for each category, as well as an additional ASN.1 sequence that contains one member for each of these categories. E.g.:
OperationExchangeItem-IN ::= SEQUENCE {
element SomeType
}

*OperationExchangeItem ::= SEQUENCE {
 iN-**Parameters** OperationExchangeItem-IN
}*

- *Merge GUIs within a single node*
Whether or not to merge all GUIs within a single node into a single master GUI. The merge will not be attempted if it would lead to change in the number of interfaces.
- *Path to hardware libraries:*
Path to AADL definitions of available components.
- *Include comments in ASN.1 definitions:*
Transfer Capella element summaries and descriptions into ASN.1 comments of the relevant ASN.1 data types.
- *Use tabs for indent in ASN.1 definitions:*
Indent ASN.1 definitions using tabs instead of spaces.
- *Member name for class realizations:*
ASN.1 sequence member name used for class realizations. E.g.:

*VehicleType ::= SEQUENCE {
 realization CHOICE {
 descendant1 CarType,
 descendant2 SpacecraftType
 }
}*

- *Member name for union values:*
ASN.1 sequence member name used for union realizations. E.g.:

*SomeUnionType ::= SEQUENCE {
 value CHOICE {
 variant1 SomeType,
 variant2 SomeOtherType
 }
}*

- *ASN.1 identifier name prefix:*
Optional prefix which may be appended to ASN.1 identifier names. E.g.:

*SomeStruct ::= SEQUENCE {
 mMemberIdentifier1 SomeType,
 mMemberIdentifier2 SomeOtherType
}*

- *ASN.1 type name prefix:*
Optional prefix which may be appended to ASN.1 type names. E.g.:

***T**SomeStruct ::= SEQUENCE {
 memberIdentifier1 **T**SomeType,
 memberIdentifier2 **T**SomeOtherType
}*

- *Package prefix:*
Optional prefix which may be appended to package names. E.g.:
***P**ExampleDataPackage DEFINITIONS AUTOMATIC TAGS ::= BEGIN*
- *Add prefix for ASN.1 identifier names if necessary:*
Append the selected prefix to an ASN.1 identifier name if the given name conflicts with a reserved word.

- *Add prefix for ASN.1 type names if necessary:*
Append the selected prefix to an ASN.1 type name if the given name conflicts with a reserved word.
- *Add prefix for package names if necessary:*
Append the selected prefix to a package name if the given name conflicts with a reserved word.
- *Force prefix for ASN.1 identifier names:*
Append the selected prefix to all identifier names.
- *Force prefix for ASN.1 type names:*
Append the selected prefix to all type names.
- *Force prefix for package names:*
Append the selected prefix to all package names.
- *Resolve ASN.1 identifier name conflicts via postfixes:*
If there is a naming conflict between 2 or more identifier names, resolve it by adding postfixes.
- *Resolve ASN.1 type name conflicts via postfixes:*
If there is a naming conflict between 2 or more type names, resolve it by adding postfixes.
- *Generate TASTE artefacts after export:*
Automatically run the update DataView and generate code skeletons commands after a physical architecture export.
- *Show TASTE incompatibility warning on Windows:*
Show warning regarding TASTE incompatibility with Windows when a Linux only command is executed on a platform detected as Microsoft Windows.
- *TASTE build command:*
Command to be invoked to build a TASTE project.
- *TASTE clean command:*
Command to be invoked to clean a TASTE project.
- *TASTE init project command:*
Command to be invoked to initialize a TASTE project.
- *TASTE edit project command:*
Command to be invoked to edit a TASTE project.
- *TASTE edit function command;:*
Command to be invoked to edit a TASTE function code.
- *TASTE generate skeletons command:*
Command to be invoked to generate code skeletons for a TASTE project.
- *TASTE update DataView command:*
Command to be invoked to update a TASTE project's DataView.aadl file.

6. Usage

6.1. Data Model

Capella-TASTE Plugin provides a new command – “Export to TASTE” – which can be used to export the selected Capella data model to ASN.1. It is available both in a dedicated menu in the menu bar (as illustrated in Figure 2) and in the context menu for the Capella Project Explorer (as illustrated in Figure 3).

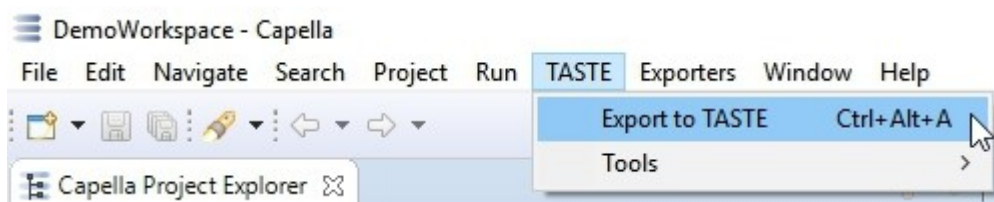


Figure 2 - Export to TASTE command in the menu bar

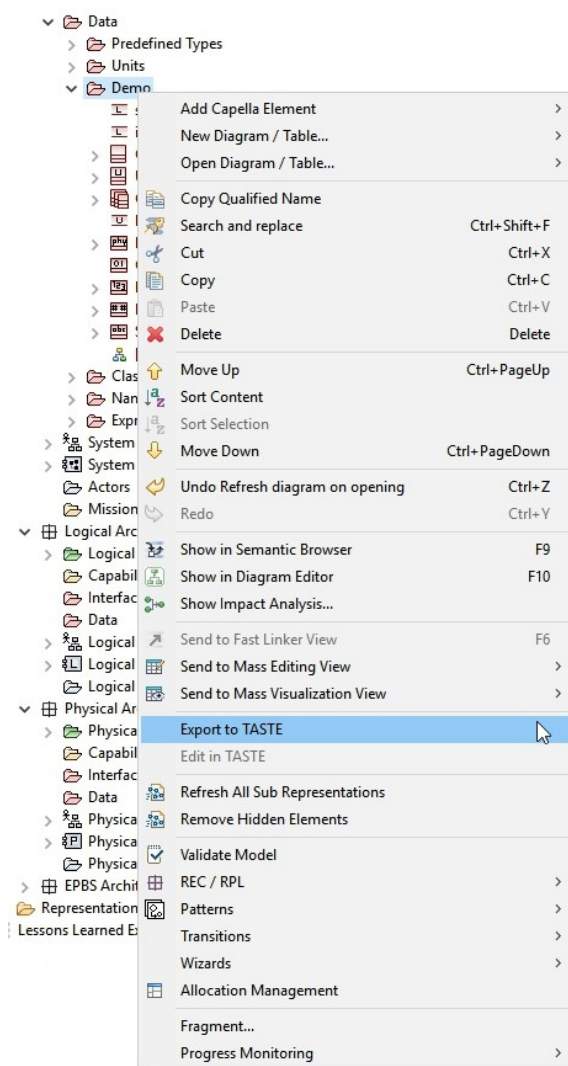


Figure 3 - Export to TASTE command in the Capella Project Explorer

For the purpose of demonstration, a simple two-package data model (illustrated in Figure 4) will be used.

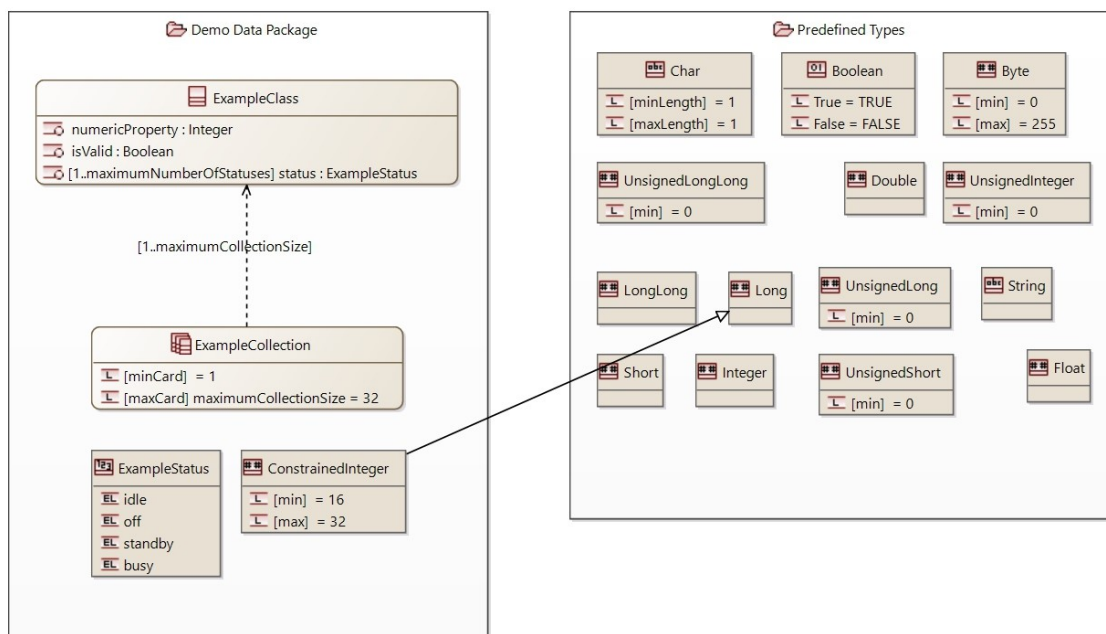


Figure 4 - Sample data model

In order to export a data model, the user should select (e.g. in the Capella Project Explorer) the data packages to be exported and then invoke the “Export to TASTE” command. If correct selection was made, then the plugin will convert the selected subset of the Capella’s data model into its own internal representation. If any issue is detected during this process, then the user is notified through a dedicated window listing all detected issues together with their severities (as illustrated in Figure 5).

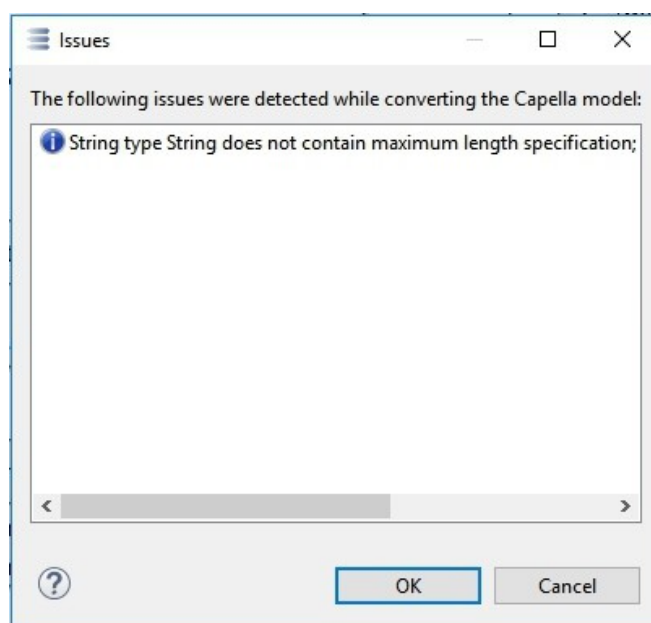


Figure 5 - Window presenting the detected issues

If the user acknowledges the listed issues, or no issues are found, then the plugin will present the list of data types and values ready for export, grouped by packages (as illustrated in Figure 6).

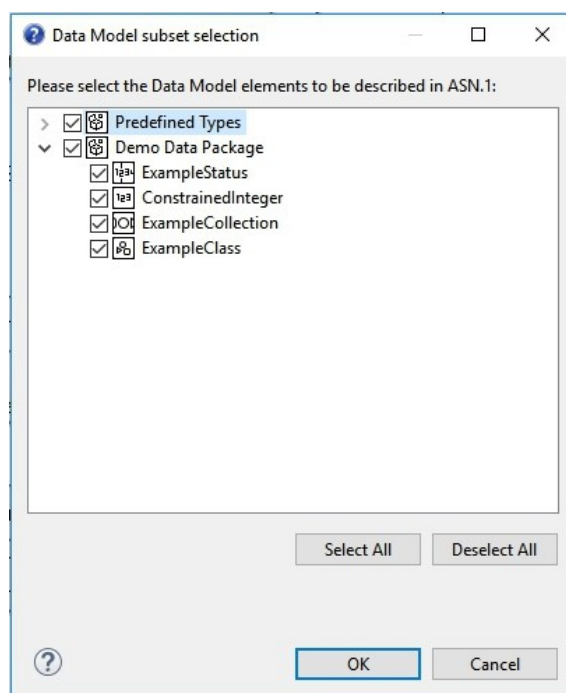


Figure 6 - Data model subset selection window

The user may adjust the initial selection, e.g. by disabling the conversion of some data types or values. After the data model subset selection is confirmed, the user is asked to select the output directory for the ASN.1 model. Upon output directory selection, the data model is exported to ASN.1 files, one for each of the selected data packages (as illustrated in Figure 7 and Figure 8).



Figure 7 – Example of exported ASN.1 files

```
Demo-Data-Package DEFINITIONS AUTOMATIC TAGS ::= BEGIN

IMPORTS
    TBoolean,
    TLong,
    TInteger
    FROM Predefined-Types
;

ExampleStatus ::= ENUMERATED {
    idle,
    off,
    standby,
    busy
}

ConstrainedInteger ::= TLong(16..32)

ExampleClass ::= SEQUENCE {
    numericProperty TInteger,
    isValid TBoolean,
    status SEQUENCE(SIZE(1..8)) OF ExampleStatus
}

ExampleCollection ::= SET(SIZE(1..32)) OF ExampleClass

END
```

Figure 8 - ASN.1 generated from the example data package

6.2. Architecture

Capella-TASTE Plugin provides a new command – “Export to TASTE” – which can be used to export the selected Capella Physical Architecture model to AADL. It is available both in a dedicated menu in the menu bar (as illustrated in Figure 2) and in the context menu for the Capella Project Explorer (as illustrated in Figure 9). It must be noted that Physical Architecture needs to be selected in the Capella Project Explorer, not a Physical System or a diagram.

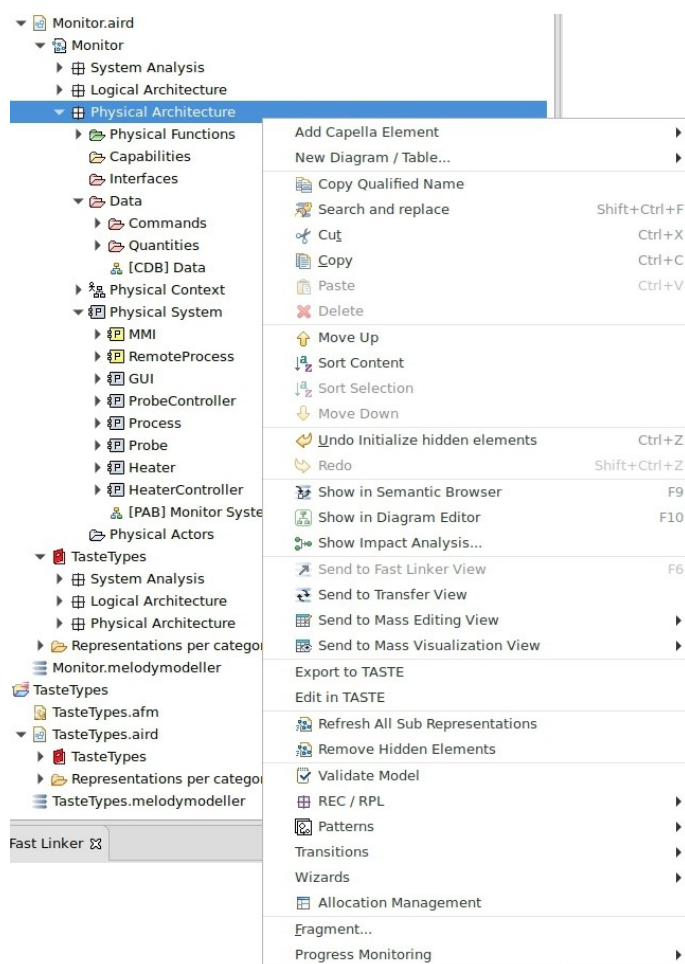


Figure 9 - Physical architecture export

The export process is the same as for ASN.1 – the user is presented with the detected issues, subset selection (as illustrated in Figure 10) and output directory selection. For each physical architecture, the following files are exported:

- DataView.asn – based on Exchange Item allocation to Functional Exchanges;
- InterfaceView.aadl – based on Physical Functions and Functional Exchanges;
- DeploymentView.aadl – based on Node Physical Components, Physical Links/Paths and Physical Function allocation.

These files require the exported ASN.1 data model. Editing in TASTE requires the DataView.asn and data model's ASN.1 files to be converted to DataView.aadl using “taste-update-data-view” command. Details are described in Chapter 6.4.

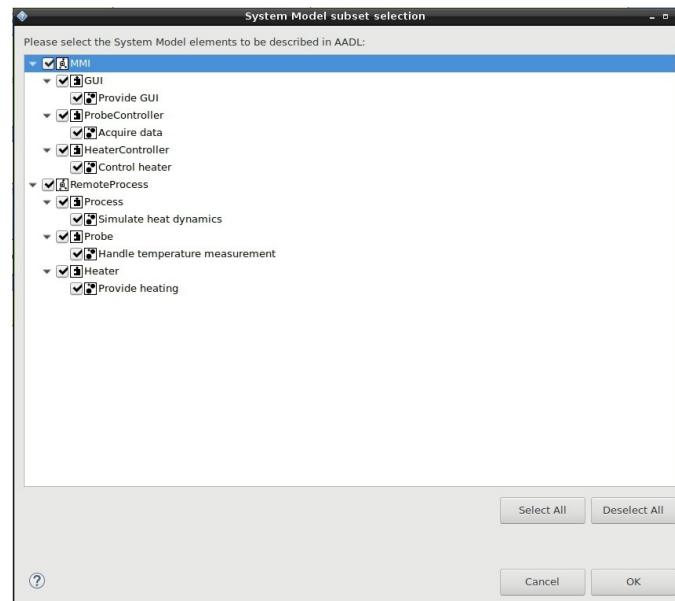


Figure 10 - Architecture subset selection

In order for the architecture to be exportable, it needs to fulfil the following requirements:

- each Physical Function should be allocated to a Behaviour Physical Component;
- each Behaviour Physical Component should be deployed onto a Node Physical Component;
- each Physical Function should have at least one Functional Exchange;
- each Functional Exchange should have at least one Exchange Item assigned;
- all Exchange Items assigned to the same Functional Exchange should have the same Exchange Mechanism set;
- each Functional Exchange should have at most one Exchange Item with Operation Exchange Mechanism;
- an Exchange Mechanism set for an Exchange Item should be one of Operation, Flow or Event;
- all Functional Exchanges between Physical Functions residing on different Node Physical Components should be allocated to Component Exchanges;
- all Component Exchanges between Behaviour Physical Components deployed on different Node Physical Components should be allocated to Physical Links or Physical Paths;

The above formalization allows to translate:

- Capella Physical Functions into TASTE Systems;
- Capella Functional Exchanges into TASTE Subprograms, Features and Connections;
- Capella Node Physical Components into TASTE Systems with Subcomponents and Partitions;
- Capella Physical Links and Physical Paths into TASTE Busses and Connections;
- Physical Ports into TASTE Devices.

The Physical Functions and Functional Exchanges serve as a basis for TASTE Interface View.

Node Physical Components and Physical Links and Paths serve as a basis for TASTE Deployment View.

Behaviour Physical Components are not translated into any TASTE concept, they serve only as allocation points for Physical Functions.

Component Exchanges between Behaviour Physical Components deployed onto a single Node Physical Component are not taken into account as they are not translated into any TASTE concept; they are also not analysed and no warnings are issued for them, as it is assumed that they may serve to express some high level design concepts.

Exchange Items allocated to Component Exchanges, Physical Links and Physical Paths are not analysed, as they are not translated into any TASTE concept and it is assumed that they may server to express some high level design concepts.

Capella Physical Architecture defines abstract functions, interfaces between them and their deployment onto different nodes. In order to create an executable code, compilation targets need to be established and function behaviour needs to be defined. In order to supply the required information, String Property Values are used.

TASTE allows the user to define a function behaviour in many ways, including, but not limited to:

- SDL diagrams;
- C code;
- Ada code.

The kind of behaviour definition must be specified in Interface View and therefore the plugin requires a Taste::Language property to specify the language used for each Physical Function. The examples of possible values are listed in Table 2.

Compilation targets must be specified in Deployment View. Each TASTE system serving as a deployment node contains a processor class declaration, which indicates the compilation target. Therefore each Node Physical Component requires a Taste::Processor property to specify the aforementioned class. The examples of possible values are listed in Table 3.

The method of communication over a Bus in TASTE is defined by its class. Therefore each Physical Link and Physical Path requires a Taste::Bus property to specify the aforementioned class. The examples of possible values are listed in Table 4.

A deployment node communicates over a Bus through a Device. A TASTE Device is created from a Physical Port, and therefore each Physical Port requires the following properties to be defined:

- Taste::Device – to specify the kind of Device used;
- Taste::ConfigurationSchema – to specify the configuration schema;
- Taste::Configuration – to specify the Device's configuration.

The examples of possible values for Taste::Device are listed in Table 5. The examples of possible values for Taste::ConfigurationSchema are listed in Table 6. The examples of possible values for Taste::Configuration are listed in Table 7.

The complete list of supported TASTE properties is listed in Table 8. Please note that all properties must be used with a "Taste::" prefix.

Table 2 - Example values of Taste::Language

| Taste::Language property value | Comment |
|--------------------------------|---|
| C | Function behaviour is defined in C code |
| Ada | Function behaviour is defined in Ada code |
| SDL | Function behaviour is defined in SDL code/diagram |
| Simulink | Function behaviour is defined in a Simulink model |

Table 3 - Example values of Taste::Processor

| Taste::Processor property value | Comment |
|--|---------------------------------|
| ocarina_processors_x86::x86.linux | Target is Linux on x86 |
| ocarina_processors_arm::stm32f407_discovery.gnat2017 | Target is baremetal STM32F407 |
| ocarina_processors_arm::stm32f429_discovery.gnat2017 | Target is baremetal STM32F429 |
| ocarina_processors_leon::gr712rc.rtems51_posix | Target is RTEMS on Leon GR712RC |

Table 4 - Example values of Taste::Bus

| Taste::Bus property value | Comment |
|-------------------------------|--------------------------------------|
| ocarina_buses::ip.i | Communication over TCP/IP connection |
| ocarina_buses::serial.generic | Communication over serial connection |

Table 5 - Example values of Taste::Device

| Taste::Device property value | Comment |
|---|-----------------------------------|
| ocarina_drivers::generic_sockets_ip.pohic | Generic IP socket for POHIC |
| ocarina_drivers::generic_serial.pohiada | Generic serial device for POHIADA |
| ocarina_drivers::STM32F4_serial.pohiada | STM32 serial device for POHIADA |

Table 6 - Example values of Taste::ConfigurationSchema

| Taste::ConfigurationSchema property value | Comment |
|---|--|
| /home/taste/tool-inst/include/ocarina/runtime/polyorb-hi-c/src/drivers/configuration/ip.asn | IP socket configuration schema location for default TASTE installation |
| /home/taste/tool-inst/include/ocarina/runtime/polyorb-hi-c/src/drivers/configuration/serial.asn | Serial configuration schema location for default TASTE installation |

Table 7 - Example values of Taste::Configuration

| Taste::Configuration property value | Comment |
|--|--|
| {devname "/dev/ttyUSB0", speed b115200, bits 8} | Example configuration of a serial device |
| {devname "eth0", address "127.0.0.1", port 5556} | Example configuration of an IP socket |

Table 8 - List of supported TASTE properties

| Taste property name | Meaning | Applies to |
|---------------------|--|--------------------------------|
| Language | Function language Translates to: Source Language | Physical Function |
| Processor | Compilation target Translates to: PROCESSOR | Node Physical Component |
| Bus | Communication bus/method Translates to: BUS | Physical Link Physical Path |
| Device | Communication device Translates to: DEVICE IMPLEMENTATION DEVICE CLASS | Physical Port |

| | | |
|----------------------|--|---|
| DeviceClass | [OPTIONAL] Override of the device class implied by Taste::Device Translates to: DEVICE CLASS (override) | Physical Port |
| ConfigurationSchema | Address of an applicable ASN.1 configuration schema Translates to: Deployment::Config | Physical Port |
| Configuration | Communication device configuration Translates to: Deployment::Configuration | Physical Port |
| Version | [OPTIONAL] Communication device version Translates to: Deployment::Version | Physical Port |
| QueueSize | [OPTIONAL] Internal message queue of an interface Translates to: Taste::Associated_Queue_Size | Functional Exchange |
| MinimumExecutionTime | [OPTIONAL] Declared minimum execution time of an interface invocation Translates to: Compute_Execution_Time | Functional Exchange |
| MaximumExecutionTime | [OPTIONAL] Declared maximum execution time of an interface invocation Translates to: Compute_Execution_Time | Functional Exchange |
| Deadline | [OPTIONAL] Declared deadline for an interface invocation Translates to Taste::Deadline | Functional Exchange |
| ExchangeKind | [OPTIONAL] Override of the implied interface type, one of sporadic, protected and unprotected Translates to: Taste::RCMoperationKind (override) | Functional Exchange |
| Timer | [OPTIONAL] Timer declaration for an SDL function Translates to: TIMER SUBCOMPONENT | Physical Function (only with Taste::Language set to SDL) |
| Directive | [OPTIONAL] Additional TASTE specific directive (e.g. linker option) | Physical Function |
| Encoding | [OPTIONAL] Encoding to be used for exchanged data items. Can be Native, ACN or UPER. (override) | Functional Exchange |

The values of aforementioned TASTE properties are not hardcoded in the code and the user is free to use any value not described in this document. However, in order to generate a valid AADL, values compatible with TASTE must be used. For that purpose, TASTE documentation [RD2] needs to be consulted. In order to simplify the process, the plugin is distributed with additional TasteTypes Capella library, which contains a set of predefined properties.

6.3. Additional TASTE integration tools

TASTE provides tools for model conversion, editing and compilation. For convenience, the plugin exposes the most important commands in the menu bar, as illustrated in Figure 11:

- Generate DataView.aadl – to convert all ASN.1 files into DataView.aadl;
- Generate code skeletons – to generate code skeletons;
- Edit project in TASTE – to launch TASTE GUI for the selected physical architecture;
- Edit function in TASTE – to launch OpenGEODE editor for the selected SDL function;
- Build system – to start the project building process.

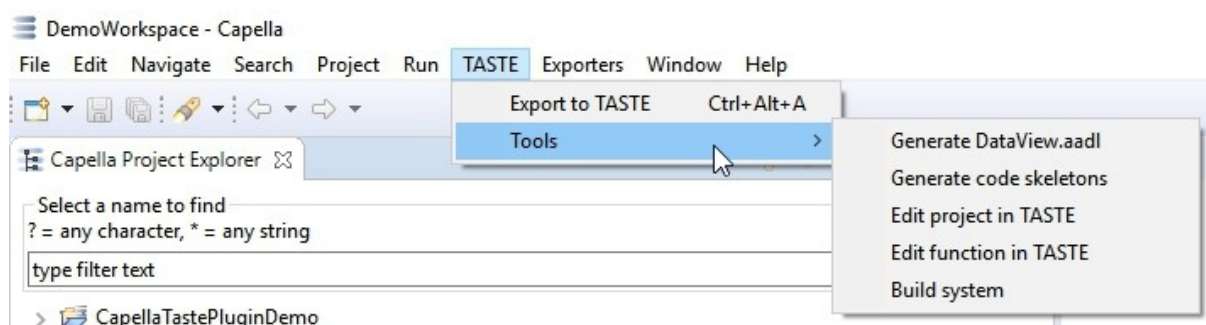


Figure 11 - Additional TASTE integration commands

It must be noted that in order to use any of these commands the physical architecture and the data model must be already exported. Code skeleton generation, project/function editing and project building will work only after DataView.aadl is successfully generated. Function editing will work only after code skeletons are generated. For more details, please refer to TASTE documentation [RD2].

All commands except “Edit function in TASTE” require the Physical Architecture to be selected in the Capella Project Explorer. “Edit function in TASTE” requires the relevant Physical Function to be selected in the Capella Project Explorer.

It must be also noted that TASTE works only in a Linux environment and so the aforementioned commands will work only on Linux.

6.4. Example workflow

The following tutorial presents the recommended workflow for using the plugin, using an example “Monitor” project. The included screenshots are uncut in order not to deprive the user of context which may be helpful in case of doubts. Each of the subsequent chapters corresponds to a single tutorial step.

6.4.1. Define the data model

First, a data model needs to be defined. In order to create a valid AADL DataView, each type must be constrained – strings must have their maximum lengths set, numeric types must be bounded, cardinalities must be finite. It is recommended not to use the PredefinedTypes, as some of them lack the required bounding. The data model should include Exchange Items. Please note that every Exchange Item must have its Exchange Mechanism set. The data model of the example project is illustrated in Figure 12.

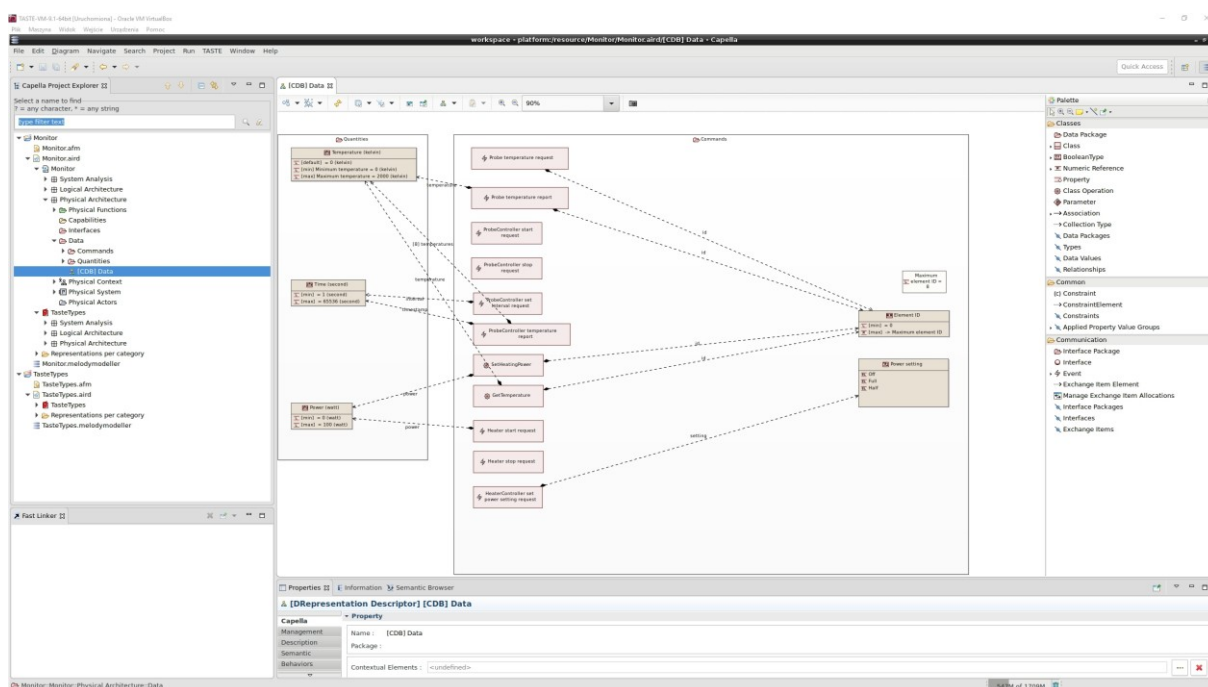


Figure 12 - Monitor data model

6.4.2. Define the architecture

When the data model is established, a physical architecture needs to be defined. It can be created either by hand, or by Capella assisted transition from a logical architecture. It must be noted that while the physical architecture can trace to the logical one, the mapping does not need to be 1:1. As TASTE Interface and Deployment Views operate on the very entities, the choice was made to create both Views from a single architecture kind. The user should ensure that all the modelling requirements described in the previous chapter are met. The physical architecture of the example project is illustrated in Figure 13. Figure 14 illustrates the assignment of Exchange Items to a Functional Exchange.

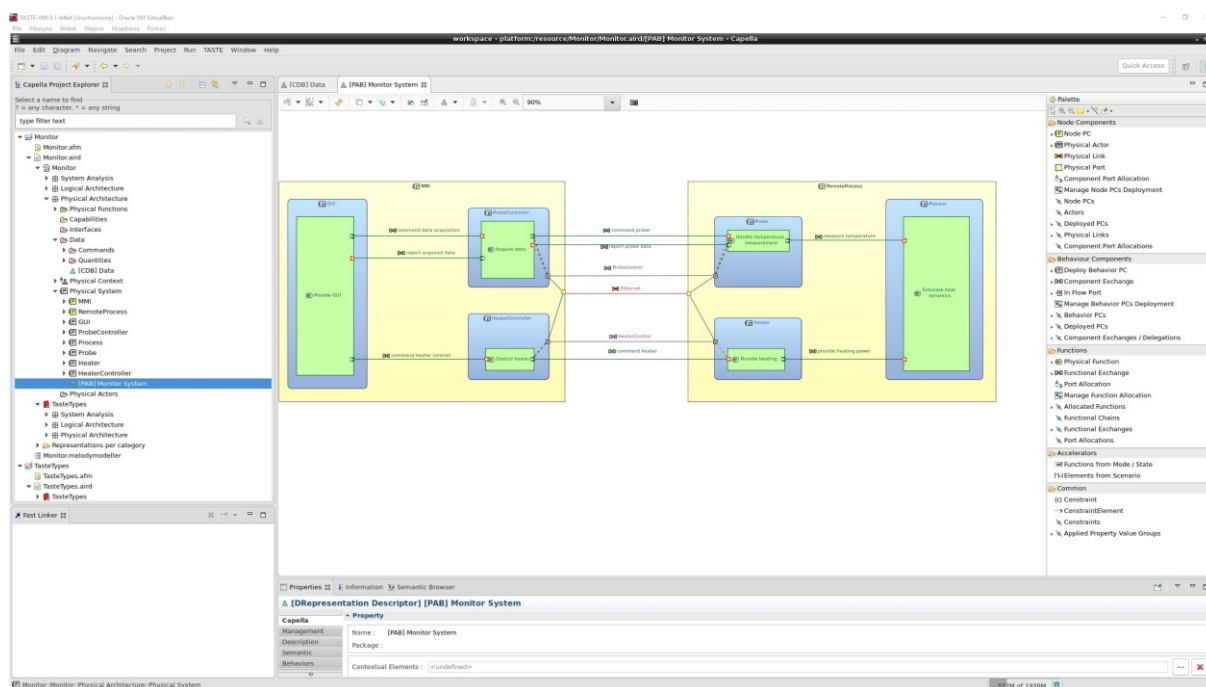


Figure 13 - Monitor physical architecture

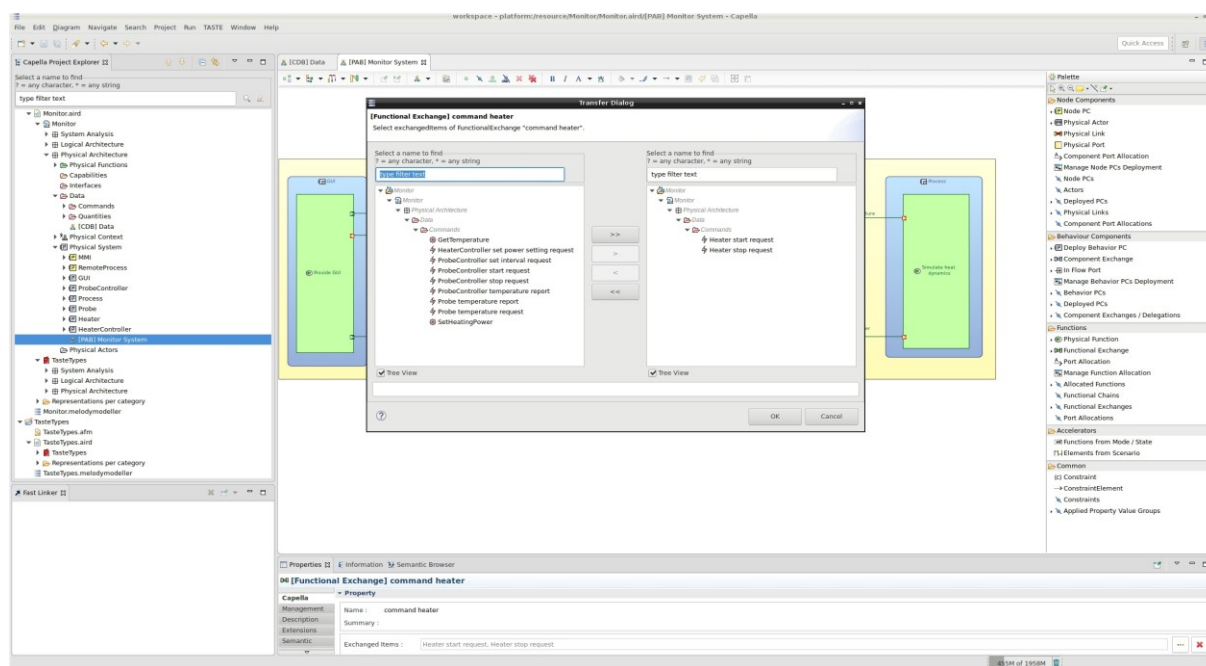


Figure 14 - Exchange Item assignment to a Functional Exchange

6.4.3. Apply the required properties

As described in the previous chapter, the required properties need to be applied to Physical Functions, Node Physical Components, Physical Links, Physical Paths and Physical Ports. In order to facilitate this process, the use of TasteTypes library is recommended. Figure 15 and Figure 16 illustrate the application of the properties.

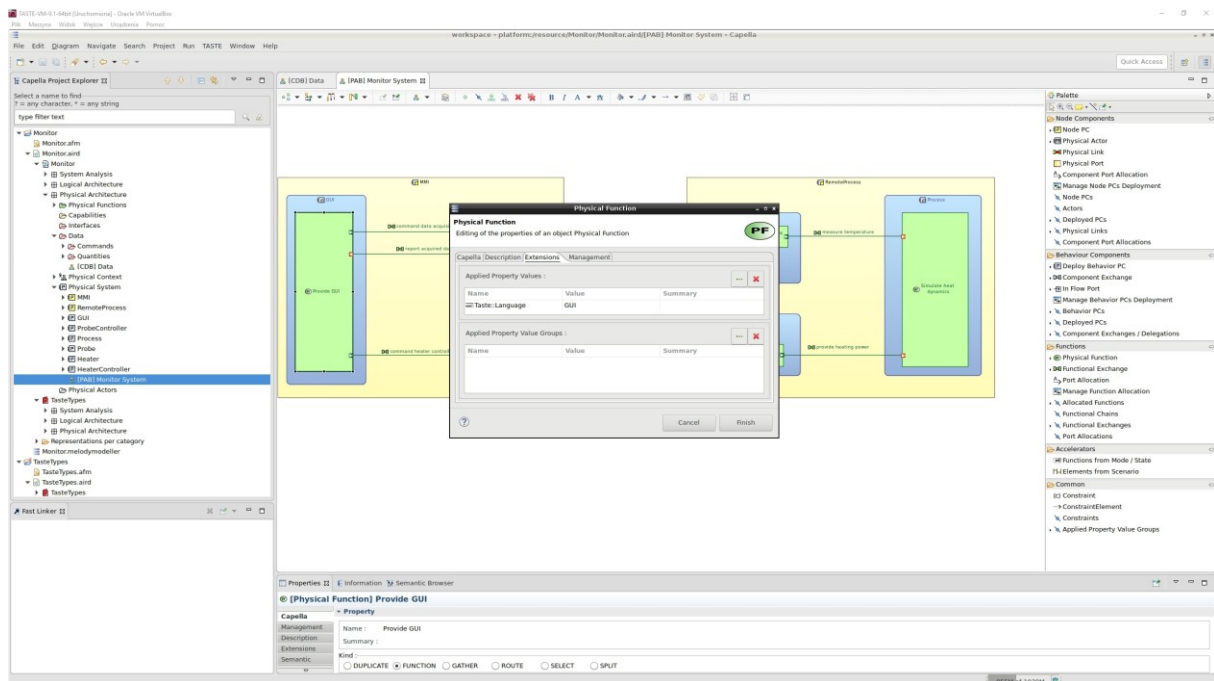


Figure 15 - Application of function properties

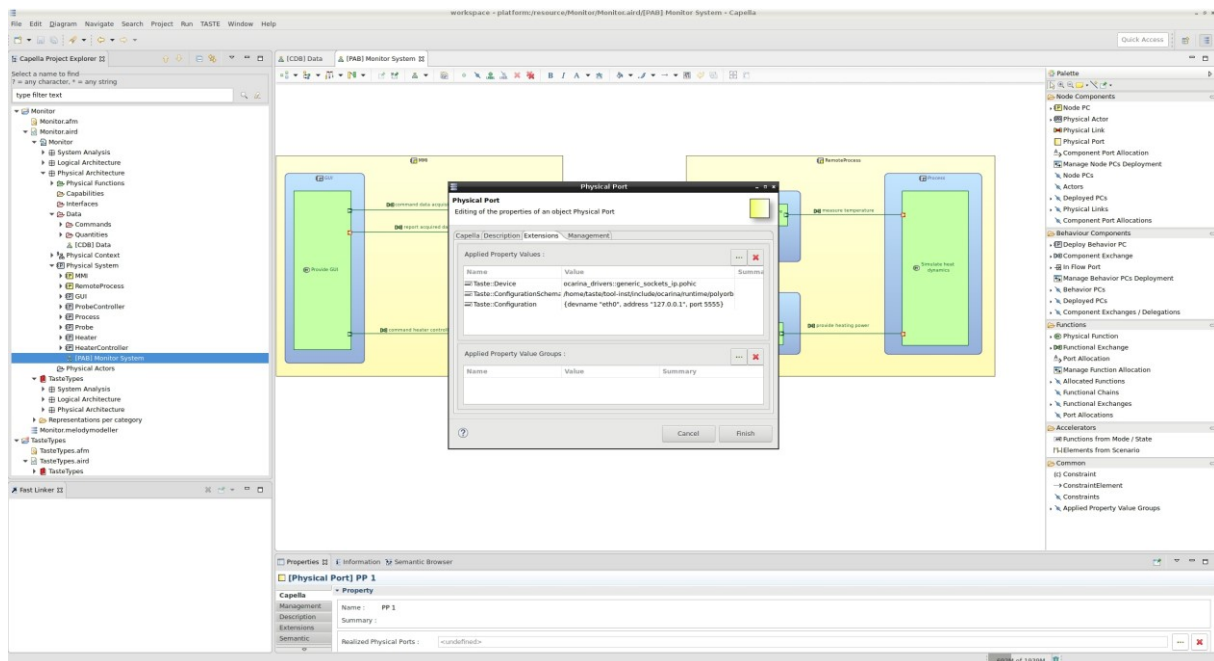
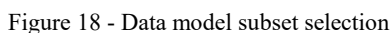


Figure 16 - Application of port properties

6.4.4. Export the data model

When ready, the data model can be exported. It is recommended to create a dedicated folder for the project artefacts. Each Package is exported to a separate file. In case of the example “Monitor” project, this is “Commands.asn” and “Quantities.asn”. After selecting the export command (Figure 17), the user must chose the subset to be exported (Figure 18). Select all.



When ready, the physical architecture can be exported. The target folder must be the same as the one for the data model. The export will create the following files: “InterfaceView.aadl”, “DeploymentView.aadl” and “DataView.asn”. These files are the same for every project and are not dependent on any project configuration. After selecting the export command (Figure 19), the user must chose the subset to be exported (Figure 20). Select all.

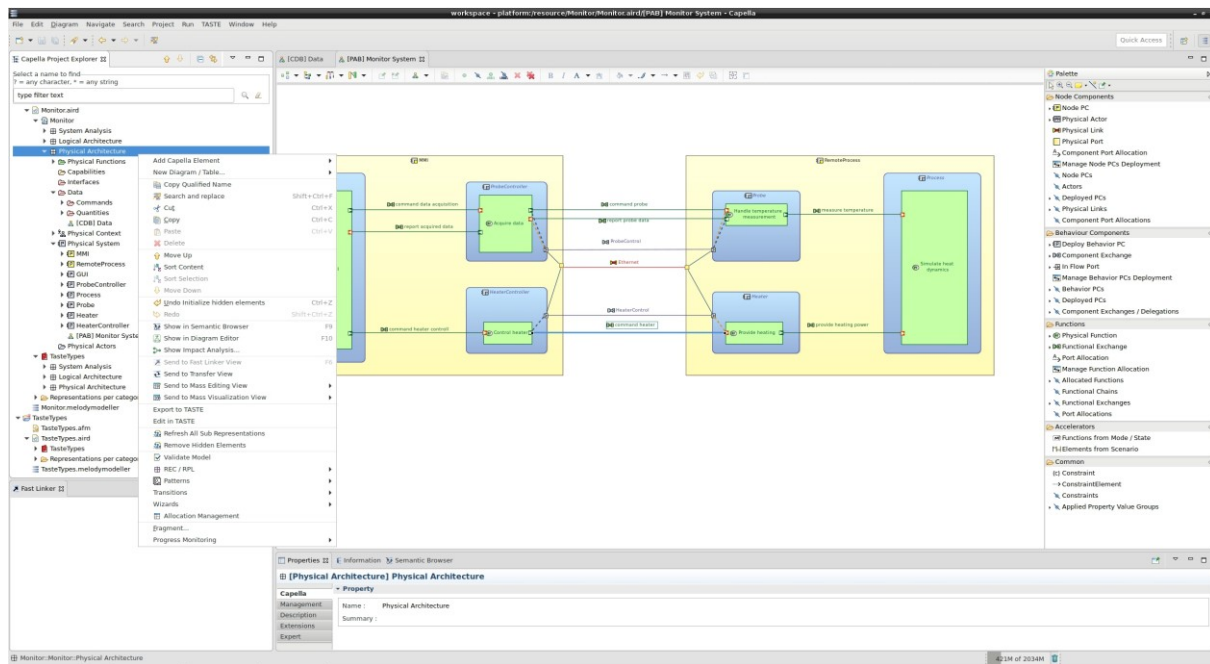


Figure 19 - Architecture export

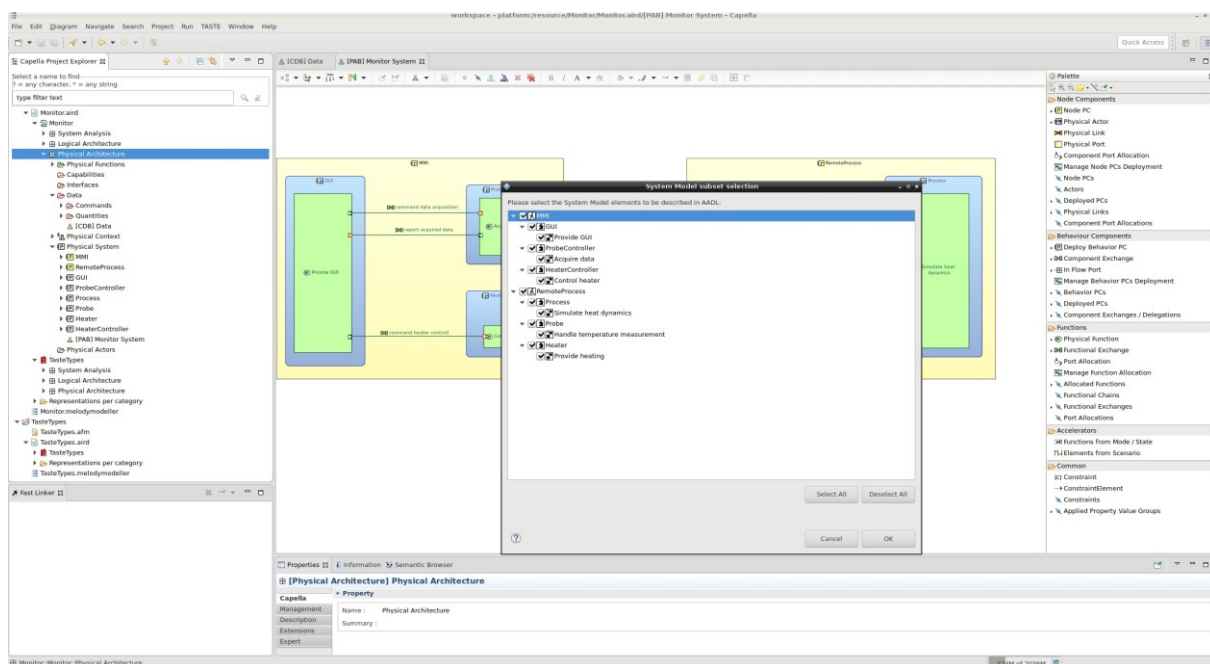


Figure 20 - Architecture subset selection

6.4.6. Postprocessing, editing and compilation

After the data model and the physical architecture are exported, the generated ASN.1 data model must be converted into AADL. This can be done using `taste-update-data-view` command or `asn2aadlPlus` directly. Please consult TASTE documentation [RD2] for details. The plugin can invoke the “`taste-update-data-view`” command directly from the menu bar, as described in Chapter 6.3.

Before a function can be edited, its skeleton must be generated in TASTE. This can be done using “`taste-generate-skeletons`” command. The plugin can invoke it directly from the menu bar.

In the default configuration, on a Linux platform, the plugin performs these two steps automatically after the physical architecture export. For the example “Monitor” project, the generated artefacts are illustrated in Figure 21.

```

acquire_data
build-script.sh
Commands.asn
control_heater
DataView.aadl
DataView.asn
debug
DeploymentView.aadl
handle_temperature_measurement
InterfaceView.aadl
InterfaceView.md5
interfaceview.pro
provide_heating
Quantities.asn
simulate_heat_dynamics
system_config.h
  
```

Figure 21 - Generated artefacts

In order to edit the TASTE project in a dedicated TASTE GUI application, “taste-edit-project” command should be used. The plugin can invoke it directly from the menu bar. The generated Interface and Deployment Views are illustrated in Figure 22 and Figure 23 respectively.

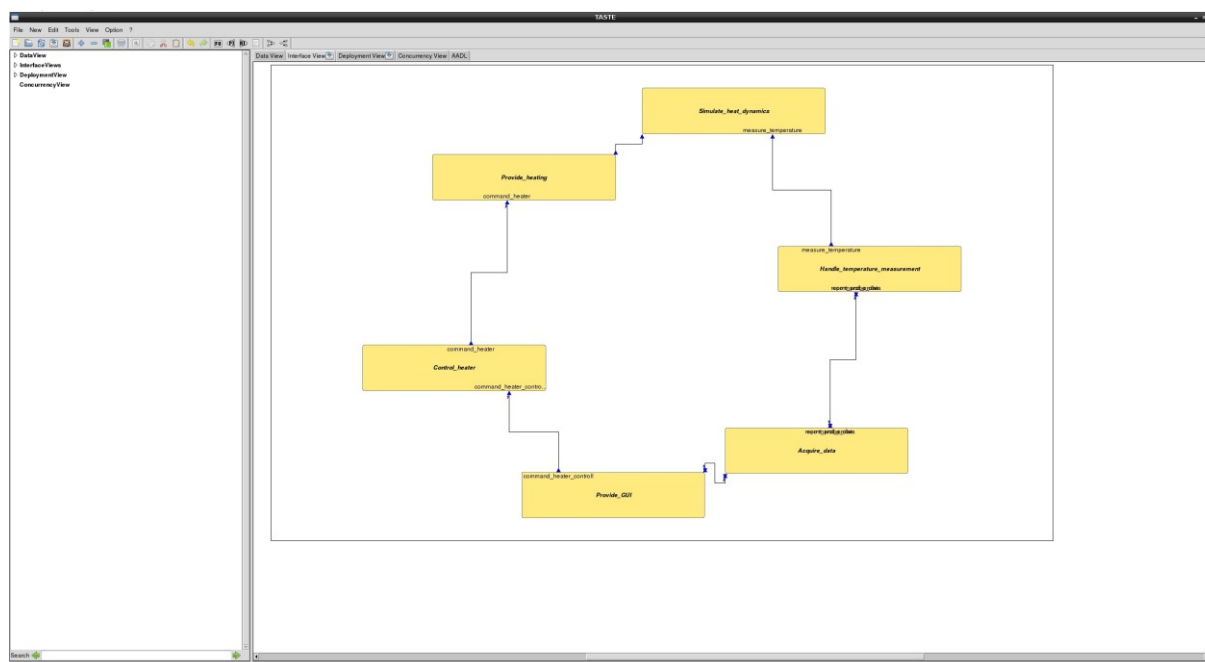


Figure 22 - Generated Interface View

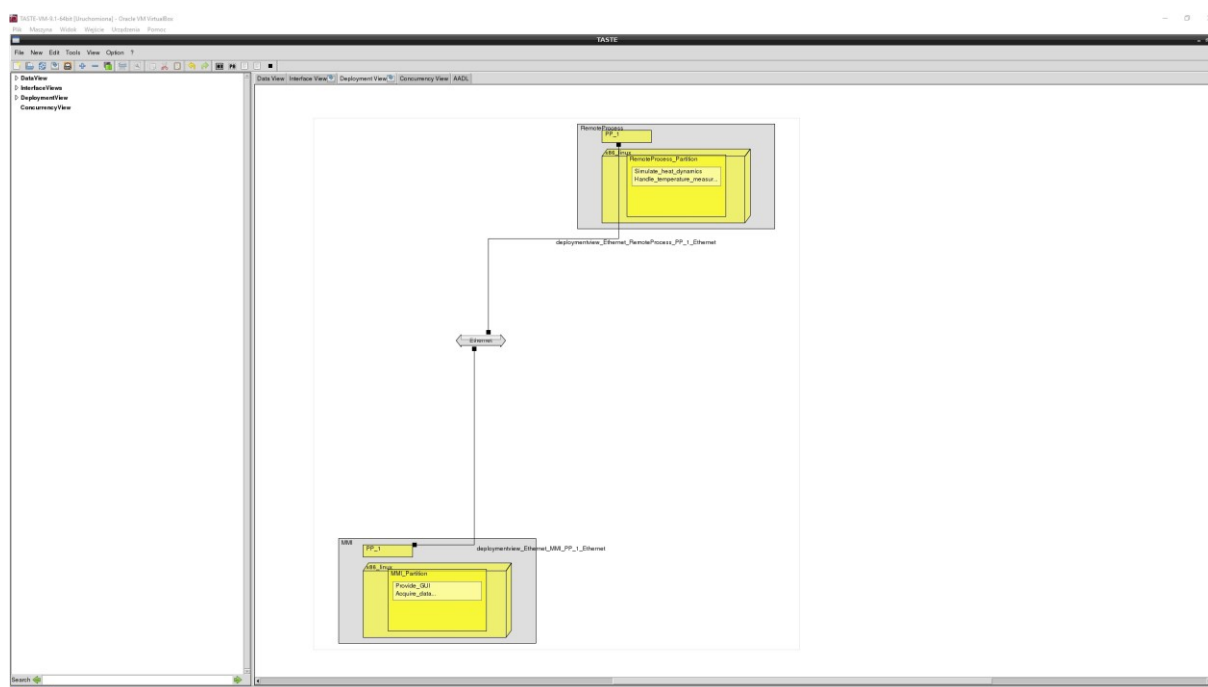


Figure 23 - Generated Deployment View

In order to compile the TASTE project into executables, “taste-build-system” command should be used. The plugin can invoke it directly from the menu bar. For the example “Monitor” project, the “Provide GUI” function with GUI as the set language results in the application illustrated in Figure 24.

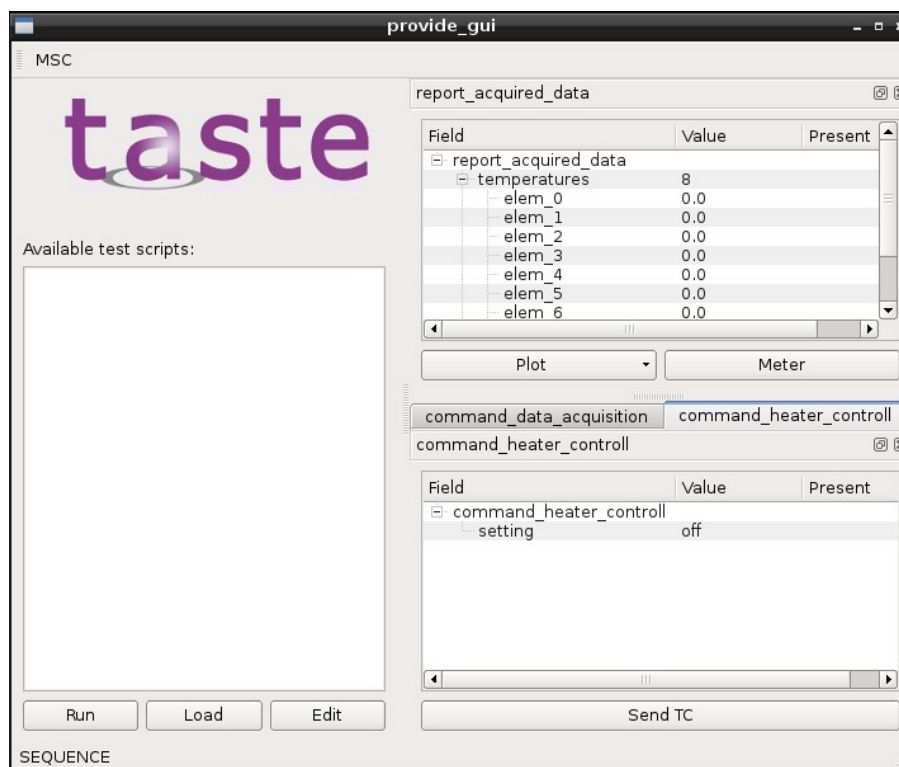


Figure 24 - Generated GUI

7. Supported Capella Features

7.1. Data Model

Capella-TASTE Plugin supports the following Capella data model constructs:

- Data Package (org.polarsys.capella.core.data.information.DataPkg)
- Boolean type (org.polarsys.capella.core.data.information.datatype.BooleanType);
- Numeric type (org.polarsys.capella.core.data.information.datatype.NumericType);
- String type (org.polarsys.capella.core.data.information.datatype.StringType);
- Enumeration type (org.polarsys.capella.core.data.information.datatype.Enumeration);
- Literal Numeric Value
(org.polarsys.capella.core.data.information.datavalue.LiteralNumericValue);
- Literal String Value (org.polarsys.capella.core.data.information.datavalue.LiteralStringValue);
- Exchange Item type (org.polarsys.capella.core.data.information.ExchangeItem);
- Collection type (org.polarsys.capella.core.data.information.Collection);
- Class and Union type (org.polarsys.capella.core.data.information.Class);
- Binary Expression (org.polarsys.capella.core.data.information.datavalue.BinaryExpression);
- Unary Expression (org.polarsys.capella.core.data.information.datavalue.UnaryExpression).

The plugin supports the concept of single parent inheritance.

“*” cardinality is supported and interpreted as “MAX”, producing valid ASN.1; this however can cause issues with the ASN1SCC compiler due to static memory allocation requirements.

As ASN.1 does not support an equivalent of expressions, the encountered binary and unary expressions are resolved, computed and translated into ASN.1 constants. Expression resolution is limited to expressions operating on integers and derivatives. The following operators are supported:

- NOT (unary): -operand
- PRE (unary): operand - 1
- SUC (unary): operand + 1
- ADD: left operand + right operand
- AND: left operand & right operand
- DIV: left operand / right operand
- EQU: left operand == right operand ? 1 : 0
- IOR: left operand | right operand
- MAX: max(left operand, right operand)
- MIN: min(left operand, right operand)
- MUL: left operand * right operand
- POW: pow(left operand, right operand)
- SUB: left operand - right operand
- XOR: left operand ^ right operand.

The expression resolver is aimed to facilitate calculating cardinalities and value ranges.

References within the interpreted types are resolved to concrete types. References are not interpreted as stand-alone entities.

7.2. System Model

Capella-TASTE Plugin supports the following Capella physical architecture constructs:

- Node Physical Component;
- Physical Actor;
- Physical Link;
- Physical Path;
- Physical Port;
- Behaviour Physical Component;
- Behaviour Physical Component deployment;
- Physical Function;
- Physical Function allocation;
- Functional Exchange;
- Functional Exchange allocation to Physical Links and Paths via Component Exchanges;
- Exchange Items exchanged by Functional Exchanges.

Node Physical Components nested within other Node Physical Components are treated the same way as Behaviour Physical Components. This distinction is not relevant to AADL generation itself, as the system subdivision into components (node or behaviour) only affects the grouping and its hierarchy presented in the system model selection dialog.

8. Best practices

8.1. Naming convention

Both ASN.1 and AADL have strict naming rules. On the other hand, Capella does not enforce any naming conventions. Therefore, in order to ensure that the generated ASN.1 and AADL contains only valid names, the plugin contains a renaming algorithm for data types. However, this algorithm consists of several basic rules and lacks the insight into the model author's intent. Therefore, for best results, it is good to adopt a naming convention inside Capella, so that the renaming is unnecessary or kept to a minimum. The recommendations for the naming convention are as follows:

- start type names with an upper-case letter;
- start member or value name with a lower-case letter;
- do not use special characters in names;
- do not start names with numbers;
- avoid using the same name in different places; in particular, do not define different data types or members with the same name;
- use unique names for physical architecture elements.

8.2. Inheritance

ASN.1 does not support the same class inheritance concept as Capella. Therefore Capella class hierarchies are flattened and translated into sequences with choices. While this solution provides data type declarations which – in many applications – can be used as classes with hierarchy, this abstraction is not transparent. Therefore, it is recommended to avoid class hierarchies if possible.

8.3. Cardinality

Due to static memory allocation requirements, the defined ASN.1 types must have fixed maximum sizes. Therefore the usage of “*” cardinality should be avoided.

8.4. Characters and strings

When defining data types from ground-up, it is possible to define a Character type first (or reuse the one from Capella's “*Predefined Types*” package), and then define various String types as collections of Characters. As IA5String is the native base type for both Strings and Characters, the optimal way to define Strings is to use the base String type and just apply the desired size constraints.

8.5. Nesting

AADL generator flattens the entire component hierarchy, as all components deployed onto a host node are transformed into a single partition. Nesting of behaviour components may make the Capella diagrams more readable and the model subset selection more efficient. However, as nested node components are treated exactly the same by the plugin as behaviour components, while still possibly making the model subset selection more efficient, they may be misleading to a diagram's viewer. Therefore it is recommended to avoid deploying node components onto other node components.

8.6. Communication layers

When designing software, it is common to dedicate some components to interfacing with external nodes via message protocols. This involves, for example, the definition of components such as “C&C

interface” or “TM/TM interface” and functions such as “Decode TC” or “Encode TM”. Given the data model and proper definitions of TASTE function parameters, TASTE automatically provides data encoding and decoding, together with data exchange handling. Therefore such components are redundant and may unnecessarily complicate the implementation. Therefore such constructs should be avoided.

8.7. GUI considerations

First versions of the plugin mapped each Capella GUI function into a separate TASTE GUI function. However, decomposition of larger, more complex projects may lead to definition of many GUI functions, each responsible for a given small task. While this may be a good logical decomposition, it may cause usability and testing issues – starting and managing many windows may be taxing for some users. Additionally, MSCs and Python scripting in TASTE are designed to work in place of a single GUI, which may make integration tests harder. Current version of the plugin presents the user with an option to merge all GUIs within each node into a single “master” GUI. Due to its convenience, it is turned on by default. However, this choice needs to be considered on a per-project basis, taking into account both the target end-user, as well as possible operational and testing scenarios.

8.8. TASTE knowledge and awareness of implementation details

System modelling in Capella is often a multi-stage process, involving possibly many people with different goals and areas of expertise. Definition of basic data types and concepts can be easily done by relevant domain experts. Definition of logical architecture may be kept abstract, without too much concern regarding the implementation details. However, as the physical architecture is translated by the plugin directly into AADL, which in turn is translated, together with the corresponding behaviour definitions, into compilable code and then executable binaries, the expert designing the physical architecture should be familiar with TASTE, its limitations and best practices. He or she should have a solid vision of the system implementation, taking into account the choice of behaviour definition languages and all the functions that the software needs to perform (even the ones not directly implied by the requirements). While it is possible to modify the resulting AADL by a TASTE expert directly in TASTE tools, it may be expensive and error prone in case of an iterative or agile approach (or even a waterfall one, due to the inevitability of changes), because, if any changes are done to the Capella model, then:

- either the changes must be manually reapplied to the generated and tweaked AADL model;
- or the AADL model needs to be regenerated and the TASTE expert’s AADL tweaks need to be reapplied again.

It may be also possible that changes to the different representations of the system may be contradictory. Therefore it is recommended not to manually edit the generated AADL and involve a TASTE expert during the physical architecture definition.

9. Messages

The list of messages generated by the plugin is presented in Table 9.

Table 9 - Messages produced by the Capella-TASTE Plugin

| Message | Recommendation |
|--|--|
| Cannot process the package. Orphan? | Verify the model consistency. |
| Type <i>\$NAME</i> has multiple super classes - multiple inheritance is not supported. | Avoid multiple inheritance. |
| Could not find package for <i>\$NAME</i> type". | Verify model consistency. |
| Cannot resolve value of element named <i>\$ELEMENT_NAME</i> of type <i>\$TYPE_NAME</i> . | The value type might be not supported by the plugin (e.g. an expression). |
| Cannot parse the value of element named <i>\$ELEMENT_NAME</i> of type <i>\$TYPE_NAME</i> . | Verify that the value is correct with respect to the value type. |
| Typed element <i>\$ELEMENT_NAME</i> does not have a type. | Verify that the given element has a type assigned. |
| Collection type <i>\$NAME</i> does not have an element type. | Verify that element type is assigned for the given collection type. |
| Critical error while trying to interpret <i>\$NAME</i> : <i>\$EXCEPTION</i> . | The plugin has encountered a critical error. Please report the issue to N7 Space. |
| Cannot extract the domain value of element named <i>\$NAME</i> . | Domain value type may be unsupported or incorrectly defined. |
| Cannot get type of element named <i>\$NAME</i> . | Verify that proper type is assigned to the given element. |
| String type <i>\$NAME</i> does not contain maximum length specification; this may cause issues with ASN.1 compilation. | Specify a finite maximum length for the given string type. |
| Function <i>\$FUNCTION_NAME</i> does not have any input ports | It is recommended for some function types (GUI and SDL) to have at least one input port. |
| Function <i>\$FUNCTION_NAME</i> does not have any output ports | It is recommended for some function types (GUI and SDL) to have at least one output port. |
| Name <i>\$NAME</i> is not unique when transformed for AADL use (<i>\$TRANSFORMED_NAME</i>) | All names used in architecture description should be unique. |
| Could not resolve the value of binary expression <i>\$EXPRESSION_NAME</i> | Verify that the expression is supported by the plugin. |
| Could not resolve the type of binary expression <i>\$EXPRESSION_NAME</i> | Verify that the expression is supported by the plugin. |
| Could not resolve the value of unary expression <i>\$EXPRESSION_NAME</i> | Verify that the expression is supported by the plugin. |
| Could not resolve the type of unary expression <i>\$EXPRESSION_NAME</i> | Verify that the expression is supported by the plugin. |
| Unary operation <i>\$OPERATION_NAME</i> is not supported | Verify that the expression is supported by the plugin. |
| Operation <i>\$OPERATION_NAME</i> could not be executed on operand <i>\$OPERAND</i> : <i>\$EXCEPTION</i> | Verify that the expression is supported by the plugin and its evaluation does not lead to numeric issues. |
| Operation <i>\$OPERATION_NAME</i> could not be executed on operands <i>\$OPERAND</i> and <i>\$OPERAND</i> : <i>\$EXCEPTION</i> | Verify that the expression is supported by the plugin and its evaluation does not lead to numeric issues (e.g. division by 0). |
| Operation not set for operands <i>\$OPERAND</i> and <i>\$OPERAND</i> | Verify that the expression has its operation correctly set. |

| | |
|--|--|
| Package for <i>\$EXCHANGE_ITEM_NAME</i> could not be found | Verify model consistency and check whether the exchange item is correctly set and defined. |
| Package for <i>\$ECHANGE_ITEM_NAME</i> could not be interpreted | Verify model consistency and check whether the exchange item is correctly set and defined. |
| Functional exchange <i>\$EXCHANGE_NAME</i> contains exchange items with different exchange mechanisms | Verify that all exchange items carried by a single exchange have the same exchange mechanisms set. |
| Functional exchange <i>\$EXCHANGE_NAME</i> does not carry any items | Ensure that all exchanges carry at least one exchange item. |
| Functional exchange <i>\$EXCHANGE_NAME</i> is an operation but the carried items are not of an operation kind | Ensure that all exchanges have mechanisms compatible with the carried exchange items. |
| Functional exchange <i>\$EXCHANGE_NAME</i> is an operation but carries multiple exchange items | Ensure that all operation exchanges carry only a single exchange item. |
| Exchange <i>\$EXCHANGE_NAME</i> is of kind <i>\$EXCHANGE_KIND</i> but is connected to a GUI function for which only sporadic kind is allowed | Ensure that all exchanges connected to a GUI function are of sporadic kind. |
| Exchange <i>\$EXCHANGE_NAME</i> is of an invalid type | Ensure that all exchanges have their mechanism set. |
| Exchange <i>\$EXCHANGE_NAME</i> is of type shared data, which is not supported | Do not use shared data as the exchange mechanism. |
| Cannot parse deadline <i>\$STRING</i> for exchange <i>\$EXCHANGE_NAME</i> | Verify that the deadline is correctly defined (a positive integer). |
| Cannot parse exchange kind <i>\$STRING</i> for exchange <i>\$EXCHANGE_NAME</i> | Verify that the exchange kind is correctly defined. |
| Operation for exchange <i>\$EXCHANGE_NAME</i> should be protected or unprotected, not sporadic | Verify that all exchange kinds are compatible with the connected functions. |
| Cannot parse maximum execution time <i>\$STRING</i> for exchange <i>\$EXCHANGE_NAME</i> | Verify that the time is correctly defined (a positive integer). |
| Cannot parse minimum execution time <i>\$STRING</i> for exchange <i>\$EXCHANGE_NAME</i> | Verify that the time is correctly defined (a positive integer). |
| Cannot parse queue size <i>\$STRING</i> for exchange <i>\$EXCHANGE_NAME</i> | Verify that the size is correctly defined (a positive integer). |
| Component <i>\$COMPONENT_NAME</i> definition cannot be found for actor <i>\$ACTOR_NAME</i> | Verify the consistency of model and its deployment links. |
| Cannot find deployment target for <i>\$COMPONENT_NAME</i> | Verify deployment links and model consistency. |
| Processor for node <i>\$COMPONENT_NAME</i> is undefined | Define processor for the given node. Only a single one can be defined. |
| Function <i>\$FUNCTION_NAME</i> defines timers but its language is not SDL | Define timers only for SDL functions. |
| Language for function <i>\$FUNCTION_NAME</i> is undefined | Define language for the given function. Only a single one can be defined. |
| Exchange <i>\$EXCHANGE_NAME</i> cannot be found in the model | Verify model consistency. |
| Source for connection <i>\$PATH_NAME</i> cannot be found | Verify model consistency. |
| Target for connection <i>\$PATH_NAME</i> cannot be found | Verify model consistency. |
| Device implementation for port <i>\$PORT_NAME</i> is undefined | Define device implementation for the given port. Only a single one can be defined. |



| | |
|--|---|
| Device class for port <i>\$PORT_NAME</i> is undefined and cannot be derived | Check device implementation definition for the given port and/or define device class. Only a single one can be defined. |
| Configuration schema for port <i>\$PORT_NAME</i> is undefined | Define configuration schema for the given port. Only a single one can be defined. |
| Configuration for port <i>\$PORT_NAME</i> is undefined | Define configuration for the given port. Only a single one can be defined. |
| Exchange <i>\$EXCHANGE_NAME</i> is between different nodes and so must be sporadic | Ensure that all exchanges between different physical nodes are sporadic and do not carry any operation exchange items. |

10. Lists

10.1. List of tables

| | |
|---|----|
| Table 1 - Capella-TASTE Plugin prerequisites | 9 |
| Table 2 - Example values of Taste::Language | 19 |
| Table 3 - Example values of Taste::Processor | 20 |
| Table 4 - Example values of Taste::Bus | 20 |
| Table 5 - Example values of Taste::Device | 20 |
| Table 6 - Example values of Taste::ConfigurationSchema | 20 |
| Table 7 - Example values of Taste::Configuration | 20 |
| Table 8 - List of supported TASTE properties | 20 |
| Table 9 - Messages produced by the Capella-TASTE Plugin | 34 |

10.2. List of figures

| | |
|--|----|
| Figure 1 - Plugin preferences page | 11 |
| Figure 2 - Export to TASTE command in the menu bar | 14 |
| Figure 3 - Export to TASTE command in the Capella Project Explorer | 14 |
| Figure 4 - Sample data model | 15 |
| Figure 5 - Window presenting the detected issues | 15 |
| Figure 6 - Data model subset selection window | 16 |
| Figure 7 - Example of exported ASN.1 files | 16 |
| Figure 8 - ASN.1 generated from the example data package | 16 |
| Figure 9 - Physical architecture export | 17 |
| Figure 10 - Architecture subset selection | 18 |
| Figure 11 - Additional TASTE integration commands | 22 |
| Figure 12 - Monitor data model | 23 |
| Figure 13 - Monitor physical architecture | 24 |
| Figure 14 - Exchange Item assignment to a Functional Exchange | 24 |
| Figure 15 - Application of function properties | 25 |
| Figure 16 - Application of port properties | 25 |
| Figure 17 - Data model export | 26 |
| Figure 18 - Data model subset selection | 26 |
| Figure 19 - Architecture export | 27 |
| Figure 20 - Architecture subset selection | 27 |
| Figure 21 - Generated artefacts | 28 |
| Figure 22 - Generated Interface View | 28 |
| Figure 23 - Generated Deployment View | 29 |
| Figure 24 - Generated GUI | 29 |