Mod Code Generator Manual

Table of Contents

1 Introduction....4
2 License....4
3 Installation....4
3.1 Python Installation....4
3.2 MCG Installation....5
3.3 Modelio Installation....6
4 MCG Overview....6
5 Modeling Guide....8
5.1 Model Creation....8
Create New Project....8
Remove Unused Elements....8
Rename Model Package....9
5.2 Type Package Creation....10
Create Type Package....10
Create Model Types....10
5.3 Model Component Creation....11
Create New Component....11
Define Component Constants....11
Create Component Operation....11
Define Operation Interface Parameters....11
Create Component Activity....13
Define Component Activity....14
Create Next Components....23
5.4 Model Package Creation....23
Create New Package....23
5.5 Modeling Rules....23
Model Rules....23
Component Attribute Rules....24
Component Operation Rules....25
Activity Diagram Rules....25
Object Node Rules....26
Operation Interaction Rules....26
Action Interaction Rules....27
Conditional Interaction Rules....27
Naming Convention Rules....28
5.6 Naming Convention....28
5.7 List of Actions....29
6 How to Run MCG....30
6.1 Converter Component (MCG CC)....30
Run Converter Component....30
Check Converter Component Outputs....31
6.2 Code Generator Component (MCG CGC)....32
Run Code Generator Component....32
Check Code Generator Component Outputs....33
7 Configuration File Syntax and Format....34

1 Introduction

The Mod Code Generator (MCG) program is a code generator tool, written in Python language, which allows to generate code from Modelio environment. You can create a model in modeling environment following section Modeling Guide and then generate C code representation of your model as explained in section How to Run MCG. A short overview of the MCG program is available in MCG Overview section.

Before you start to use the MCG program please familiarize yourself with license terms and conditions, summarized under below section License.

2 License

The MCG program is available under the terms of the GNU General Public License, either version 3 of the license, or (at your option) any later version.

Under Section 7 of GPL version 3, you are granted additional permissions described in the MCG Output Exception, version 1.

You should have received a copy of the GNU General Public License and the MCG Output Exception along with this program, look for **LICENSE** and **MCG OUTPUT EXCEPTION** files available under the MCG release.

If you wish to use the MCG program, then you shall read and comply with terms and conditions of above license and permission.

Apart from the above, the MCG program works with:

- Python, available under Python Software Foundation License Version 2,
- Modelio, available under GNU General Public License either version 3 of the license, or (at your option) any later version,

therefore you shall in addition read and comply with terms and conditions of above licenses in order to use Python and Modelio programs.

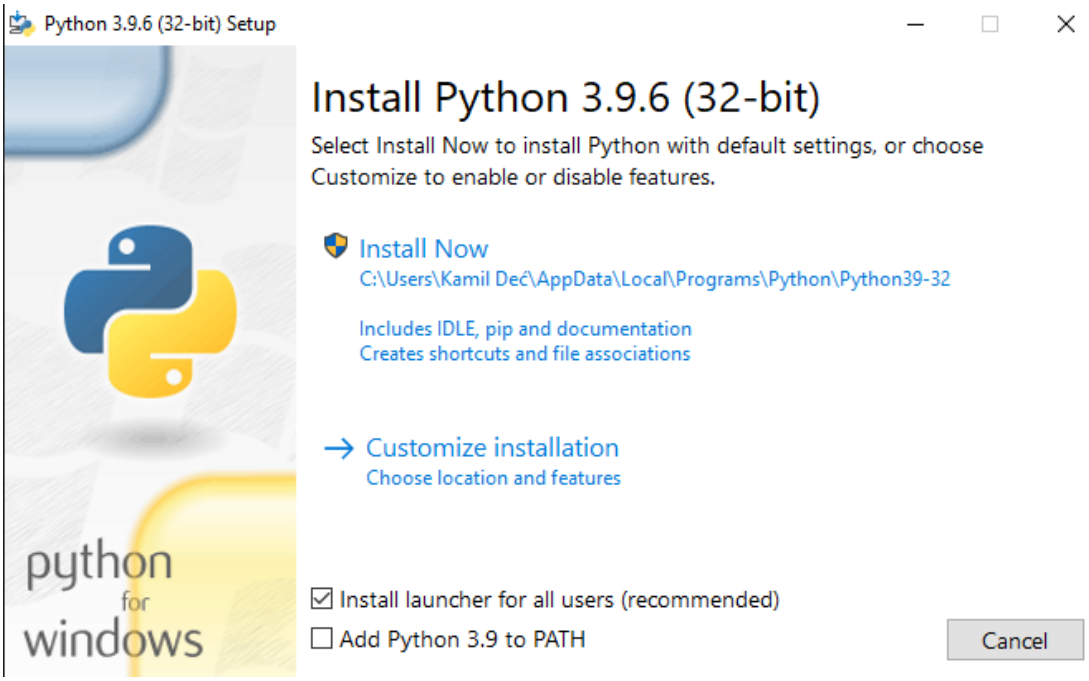
You should have received a copy of above licenses along with their installation packages, which are not enclosed directly with the MCG release. Please follow Installation to see how to obtain installation packages.

3 Installation

3.1 Python Installation

The MCG program requires Python package to run, therefore please install Python release 3.9.6 from https://www.python.org/.

During installation make sure to select **Install launcher for all users (recommended)**, because Python launcher will be used to run the MCG program with correct Python version. More information about Python installation on Windows OS and Python Launcher can be found under https://docs.python.org/3.9/using/windows.html# and https://docs.python.org/3.9/using/windows.html#launcher.



Once required Python release is installed, please open a command line window and type `py -0` to see a list of available Python releases on your computer. The list should contain "-3.9-xx", where "xx" is equal to either 32 or 64 depending whether you installed 32 or 64 bit version of Python release.

```
C:\>py -0
Installed Pythons found by py Launcher for Windows
 -3.9-64 *
 -3.9-32
```

Further please check if correct Python version is available from the command line. To do this please type `py -3.9-xx --version` in the command line window. You should receive "Python 3.9.6" in reply.

```
C:\>py -3.9-64 --version
Python 3.9.6
```

3.2 MCG Installation

To install the MCG simply copy MCG folder from this release to your hard drive and create following environment variables with paths to required MCG components (i.e. Converter Component and Code Generation Component):

Table 3.2.1: The MCG Environment Variables

Variable	Value
MCG_CC	py -3.9-xx "<mcg installation directory>\MCG_CC\ mcg_cc_main.py"
MCG_CGC	py -3.9-xx "<mcg installation directory>\MCG_CGC\ mcg_cgc_main.py"

, where "xx" is equal to either 32 or 64 depending whether you installed 32 or 64 bit version of Python release. Please open a command line window to check if created environment variables work correctly.

Type %MCG_CC% to check if the Converter Component is invoked:

```
C:\>%MCG_CC%

Mod Code Generator (MCG)
Copyright (C) 2021-2022 Kamil Deć github.com/deckamil
This is Converter Component (CC) of Mod Code Generator (MCG)
vX.Y.Z

License GPLv3+: GNU GPL version 3 or later.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Incorrect number of command line arguments, MCG CC process cancelled.
Usage: python mcg_cc_main.py "<model_dir_path>" "<output_dir_path>"
Arguments:
    <model_dir_path>       Path to model directory, where all catalogs with .exml files are stored
    <output_dir_path>      Path to output directory, where results from MCG CC will be saved

Keep specific order of arguments, as pointed in usage above.
See Mod Code Generator Manual for further details.
```

Type %MCG_CGC% to check if the Code Generator Component is invoked:

```
C:\>%MCG_CGC%

Mod Code Generator (MCG)
Copyright (C) 2022 Kamil Deć github.com/deckamil
This is Code Generator Component (CGC) of Mod Code Generator (MCG)
vX.Y.Z

License GPLv3+: GNU GPL version 3 or later.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Incorrect number of command line arguments, MCG CGC process cancelled.
Usage: python mcg_cgc_main.py "<config_file_path>" "<output_dir_path>"
Arguments:
    <config_file_path>     Path to configuration file, which contains source data to code generation
    <output_dir_path>      Path to output directory, where results from MCG CGC will be saved

Keep specific order of arguments, as pointed in usage above.
See Mod Code Generator Manual for further details.
```

3.3 Modelio Installation

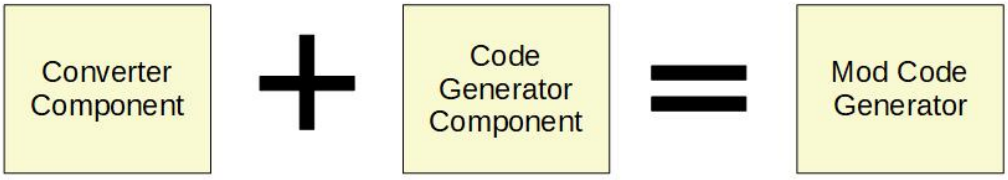
The Modelio environment is optional. If you wish to use only the CGC component of the MCG program and create configuration file on your own, then you can skip installation of this modeling environment.

However, to use full capabilities of the MCG and use both the CC and the CGC components, please install Modelio release 4.1.0 from https://www.modelio.org/.

4 MCG Overview

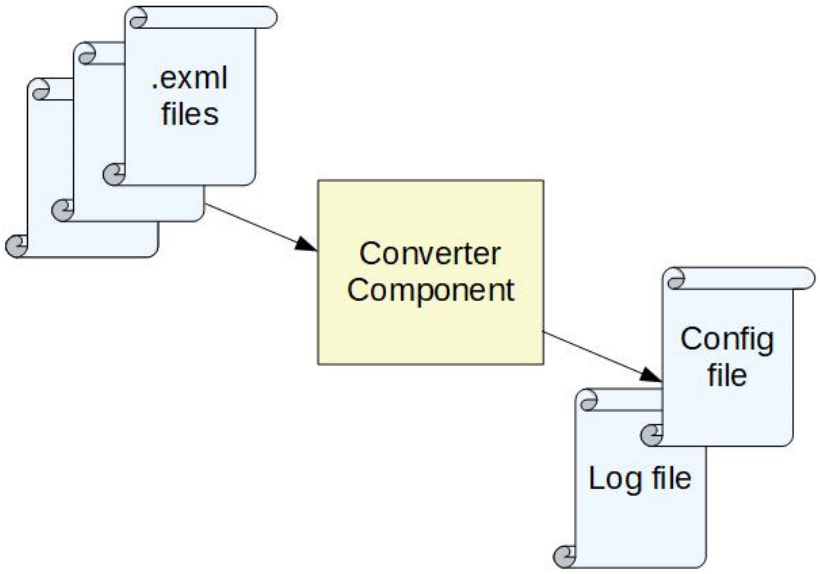
The MCG program consists of two main components (subprograms):

- the Converter Component (CC), which is responsible for conversion of a model created within the modeling environment into configuration file,
- the Code Generator Component (CGC), which is responsible for code generation from the configuration file.

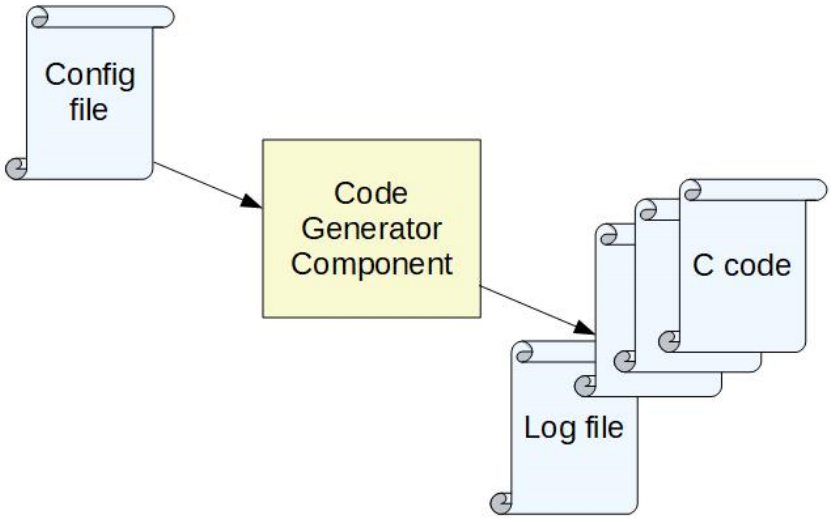


Each component is run independently. In order to run the MCG components please follow How to Run MCG section.

Model details are saved and stored in set of .exml files. The MCG CC subprogram reads model details from .exml files (model interface elements, connections between model elements, interactions between model data, etc.) and then generates the MCG CGC configuration file along with the MCG CC log file.



The MCG CGC subprogram takes the MCG CGC configuration file and base on it generates C code representation of the model and as well the MCG CGC log file. The configuration file contains details about model interfaces, data flow and data interactions in specific syntax and format, described in section Configuration File Syntax and Format.



There is a reason why the MCG CC and CGC are separate, standalone subprograms. If you wish, you can create configuration file on your own without engaging the MCG CC subprogram. You have also possibility to replace the MCG CC with your own converter subprogram, which could convert model data from different modeling environment. As long as that alternative converter will deliver configuration file in required syntax and format, the MCG CGC subprogram will be able to generate C code.

5 Modeling Guide

The modeling process will be focused on data flow through model elements and data interaction between model elements.

Components will be used to define interactions between model data and packets will be used to segregate components. Each model component is treated as a separate module and MCG will generate separate C source file for each component.

Please see below guide, which will give you introduction into modeling rules and as well please see Modeling Rules where you can find list of rules that you have to follow during modeling process.

5.1 Model Creation

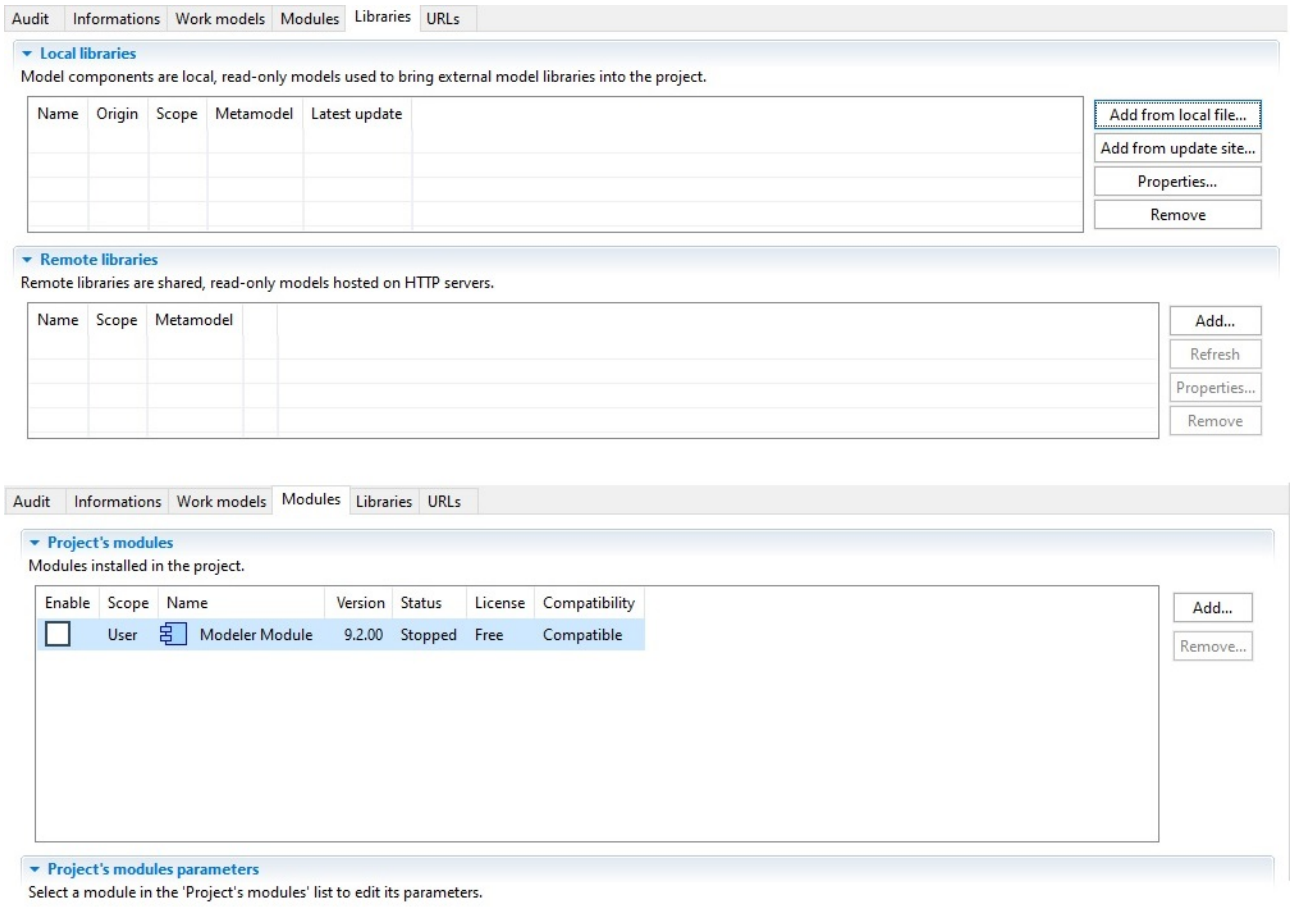
Create New Project

Open modeling environment, then select File → Switch workspace and select directory, where you would like to place new model.

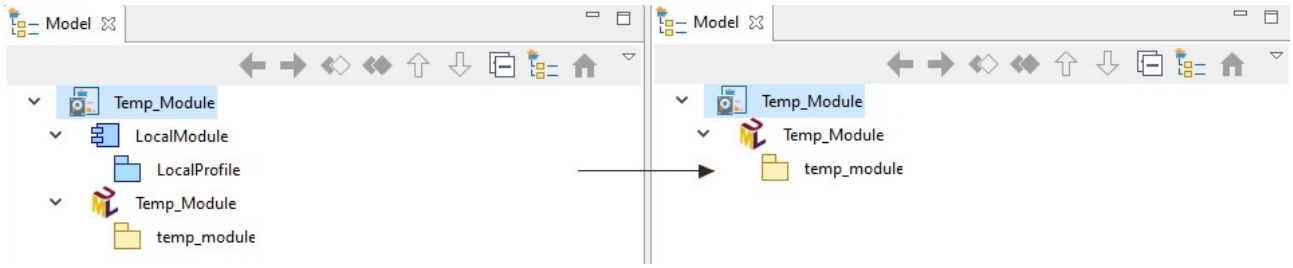
After that select File → Create a project. In Project name property type desired project name, then click Create the project.

Remove Unused Elements

Select Configuration → Libraries, then remove all Local libraries and Remote libraries. After that go to Modules tab and disable all modules, then close Project configuration window.

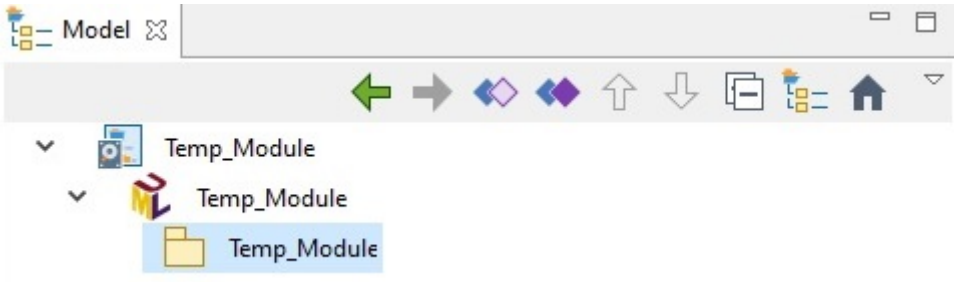


Expand all model elements in model tree. Find and remove LocalModule element.



Rename Model Package.

Locate model package under UML sub-project element, click on it and press F2 on keyboard, then type desired model package name.



5.2 Type Package Creation

Create Type Package

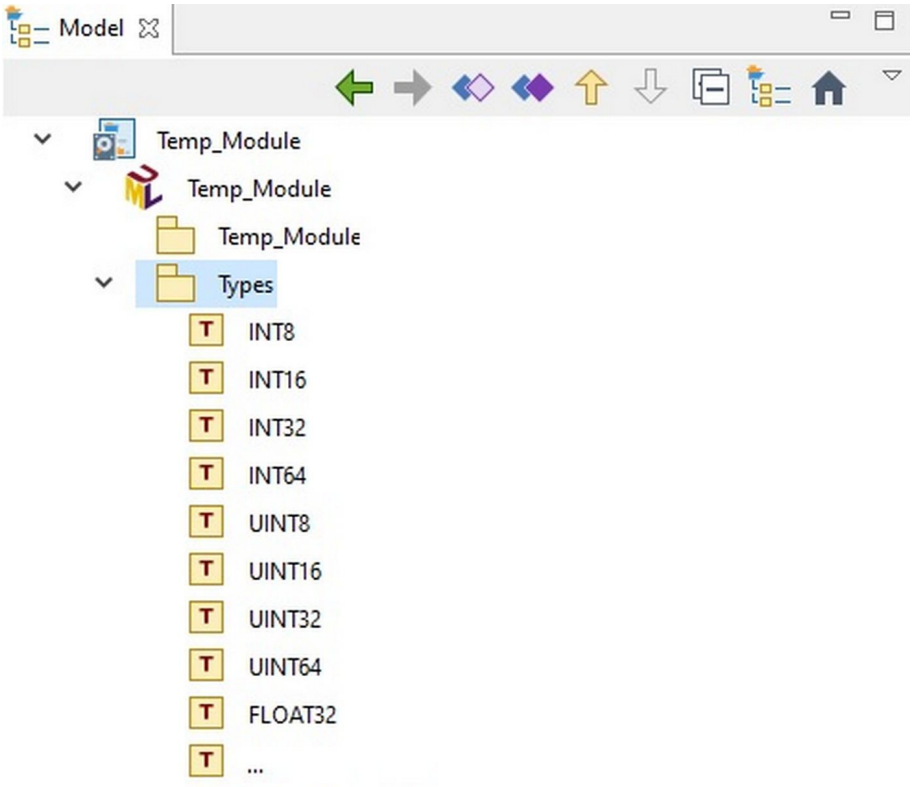
Right-click on UML sub-project element and select Create element → Package. Click on new package, press F2 on keyboard and name it as Types.

Create Model Types

Define all required data types:

- INT8
- INT16
- INT32
- INT64
- UINT8
- UINT16
- UINT32
- UINT64
- FLOAT32
- FLOAT64
- BOOL

Right-click on type package and select Create element → Data Type. Click on new data type, press F2 on keyboard and name new data type required above. Repeat the process for the rest of required data types.



5.3 Model Component Creation

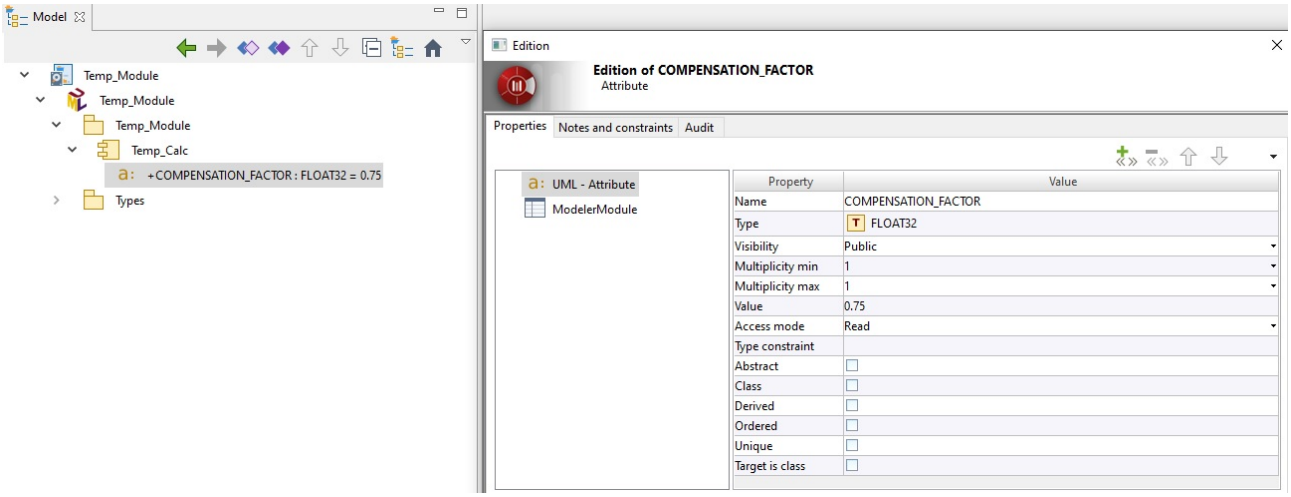
Create New Component

Right-click on model package and select Create element → Component. Click on new component element, press F2 on keyboard and then give desired name to it.

Define Component Constants

If you wish, you can define component constants and use them later on component activity diagrams. Right-click on component and select Create element → Attribute. Double-click on component attribute to open Edition window and define parameters of component constant.

In Name field type desired name of the constant and start typing desired type in Type field, then press “Enter” on keyboard to see auto-completion list (see Create Model Types section for allowed data types). In Value field set desired constant value and then set Access mode to Read.



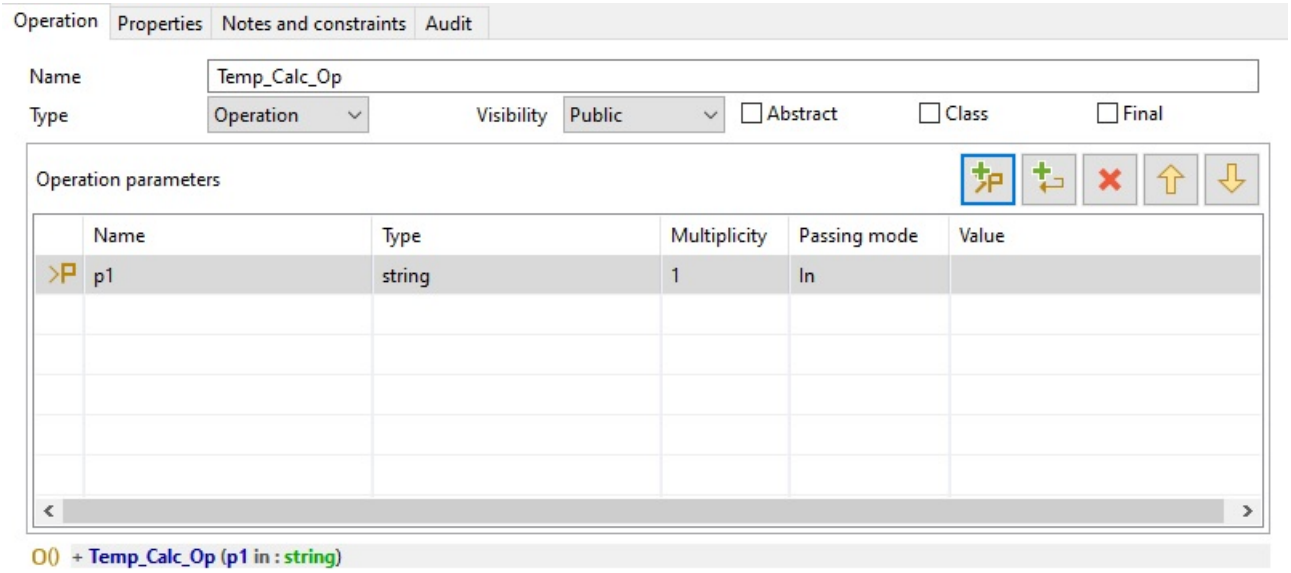
Repeat the process to add definition of other component constants if needed.

Create Component Operation

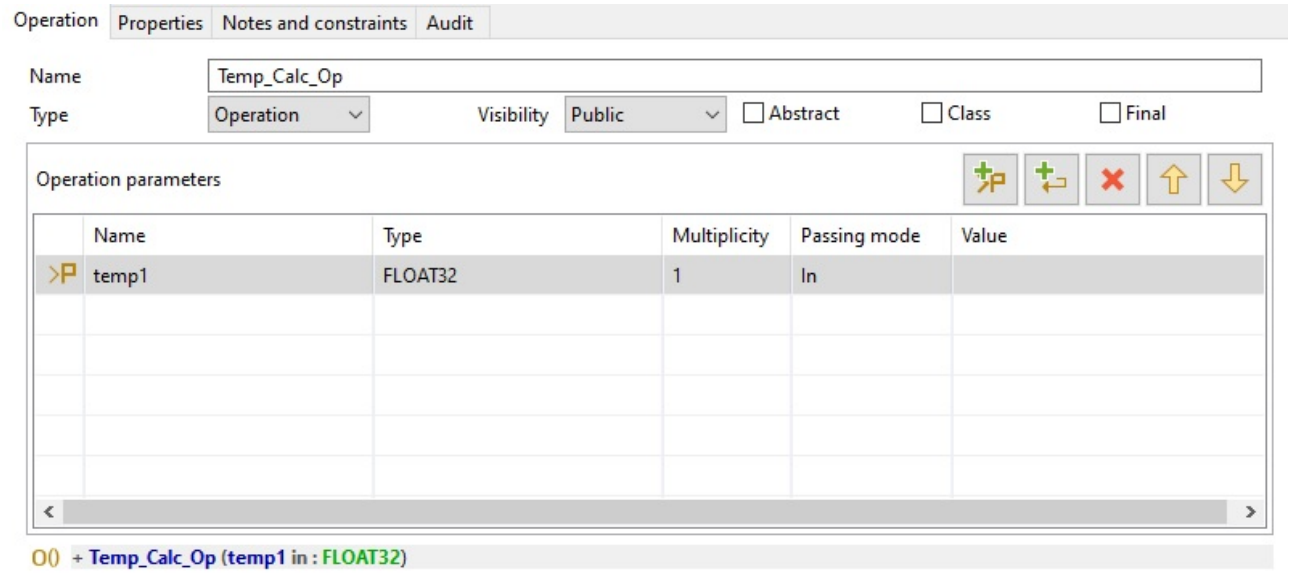
Right-click on component and select Create element → Operation. Click on new operation element, press F2 on keyboard and name new and then give desired name to it.

Define Operation Interface Parameters

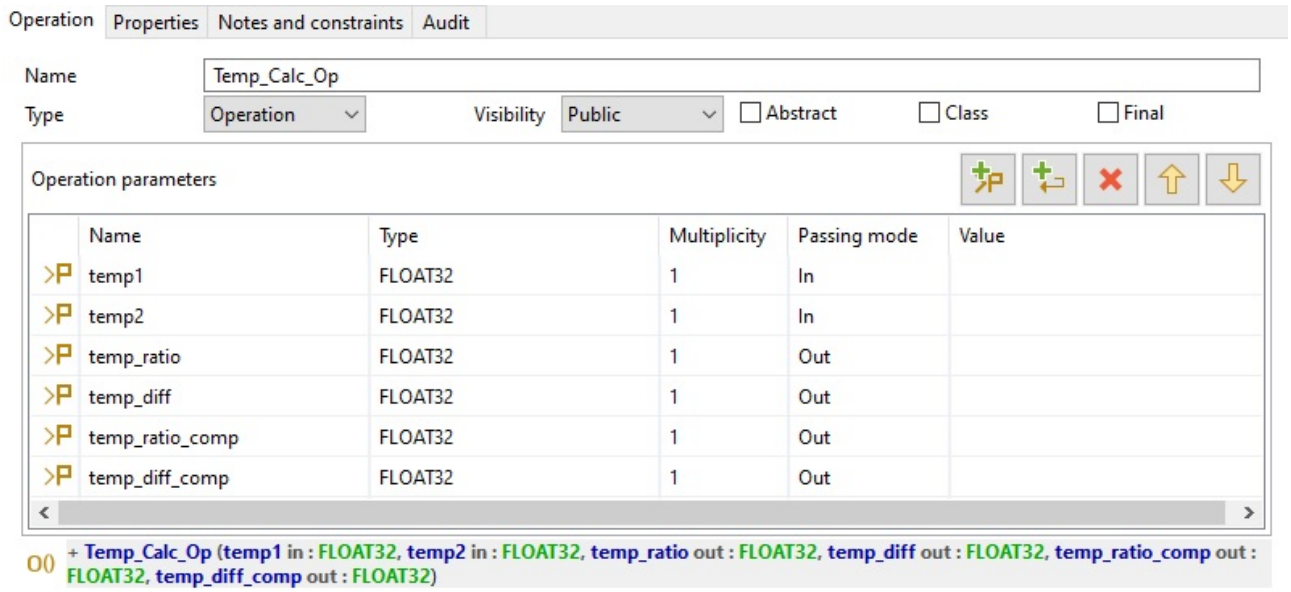
Double-click on component operation to open Edition window and define input and output parameters of the operation. Click on Create an In/Out parameter to create first parameter.



In Name field type desired name of the parameter and start typing desired type in Type field, then press “Enter” on keyboard to see auto-completion list (see Create Model Types section for allowed data types). At the end select Passing mode for your parameter: In to use it as operation input parameter and Out to use it as operation output parameter.



Repeat the process to define the rest of required parameters if needed.

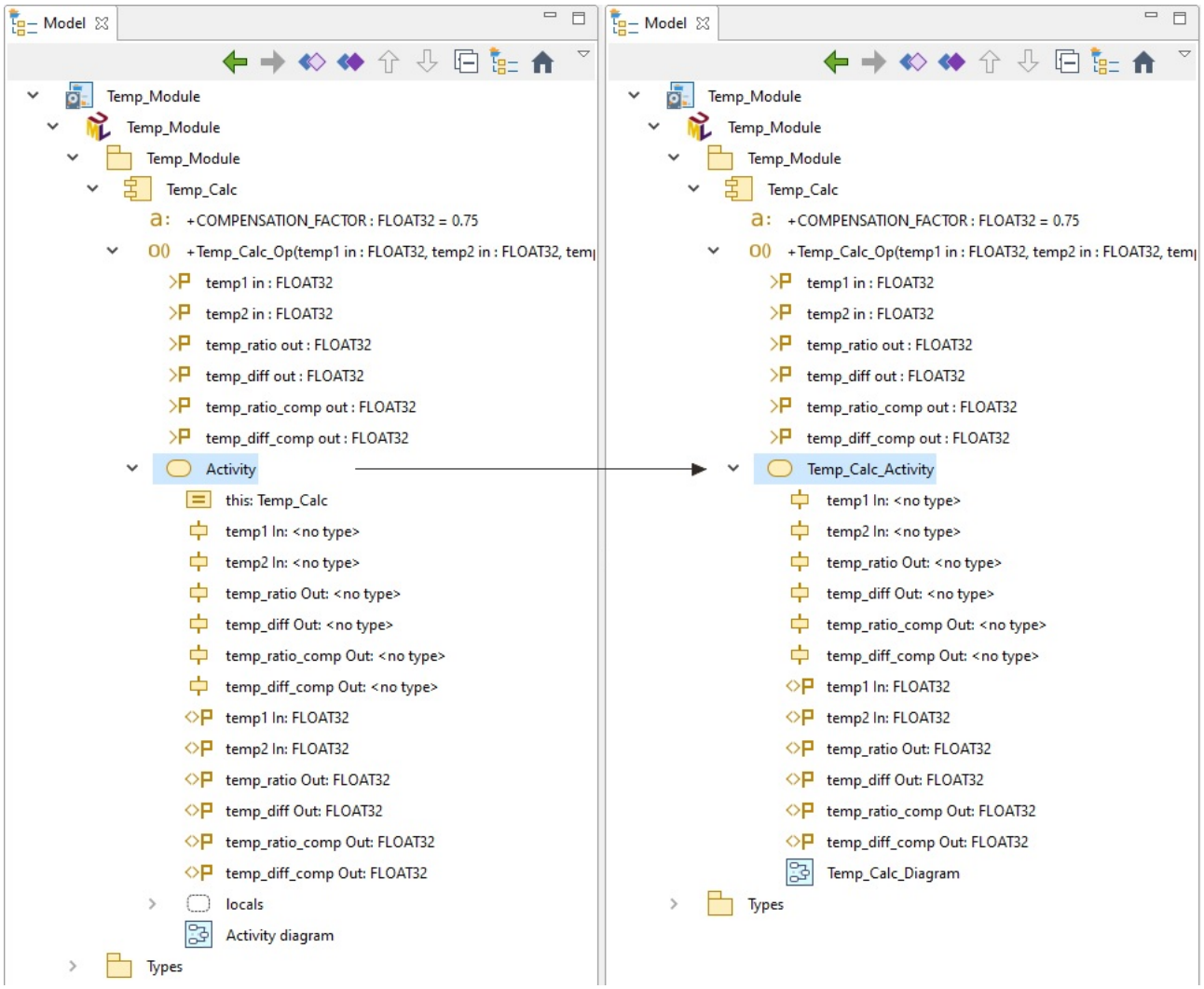


Create Component Activity

Right-click on component operation and select Create element → Activity. Unfold new activity element, then find and remove following elements under it:

1. this: <component name>
2. locals

Click on new activity element, press F2 on keyboard and then give desired name to it. In the same way rename activity diagram element that appears under activity if needed.



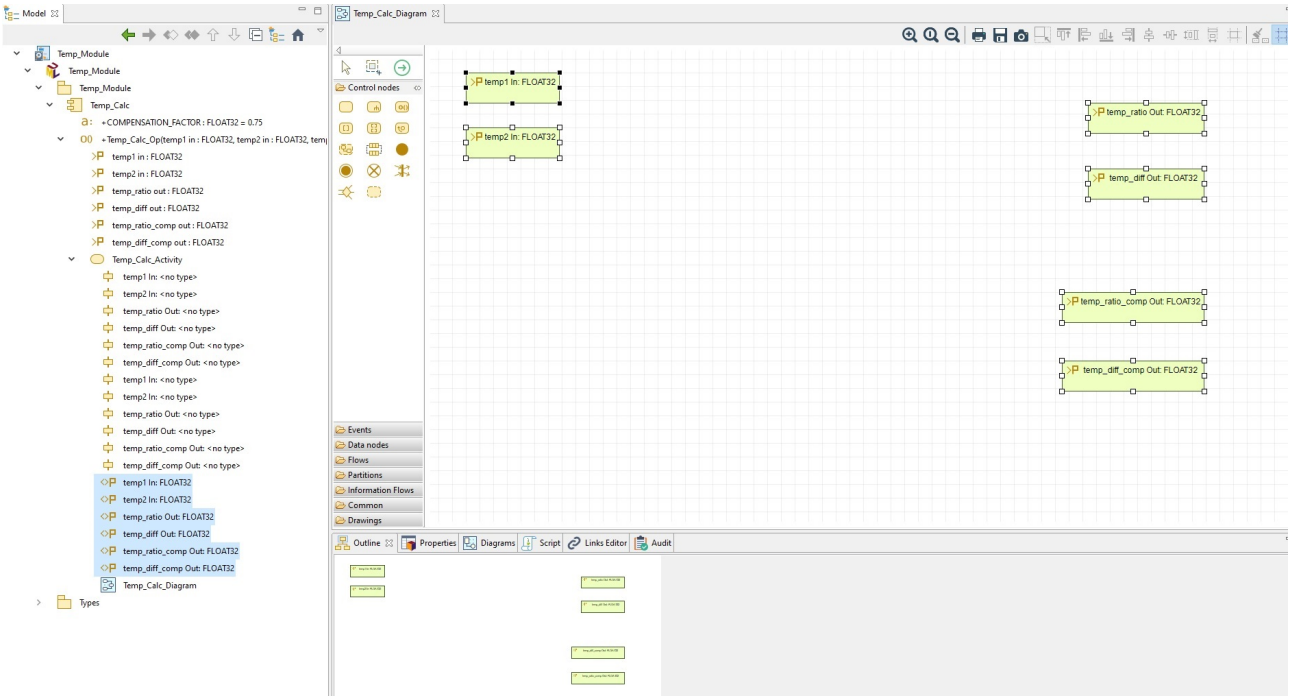
Define Component Activity

The main goal of this process is to define data and interaction elements on activity diagram and then define data flow between these diagram elements.

The activity diagram may contain interface parameters, local data elements, constants, actions, operation calls and conditional blocks.

Adding interface parameters

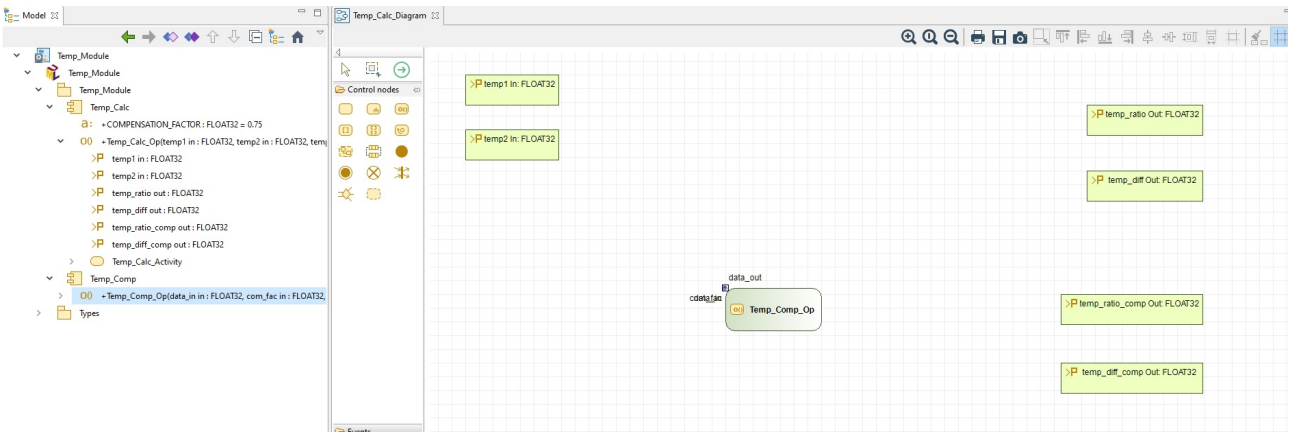
Open activity diagram element, then drag and drop interface parameters that appear under activity element into diagram space (do not use interface parameters that appear under operation element), then move them to desired position on activity diagram. Drag and drop any parameter again if you need to use it more than one time.



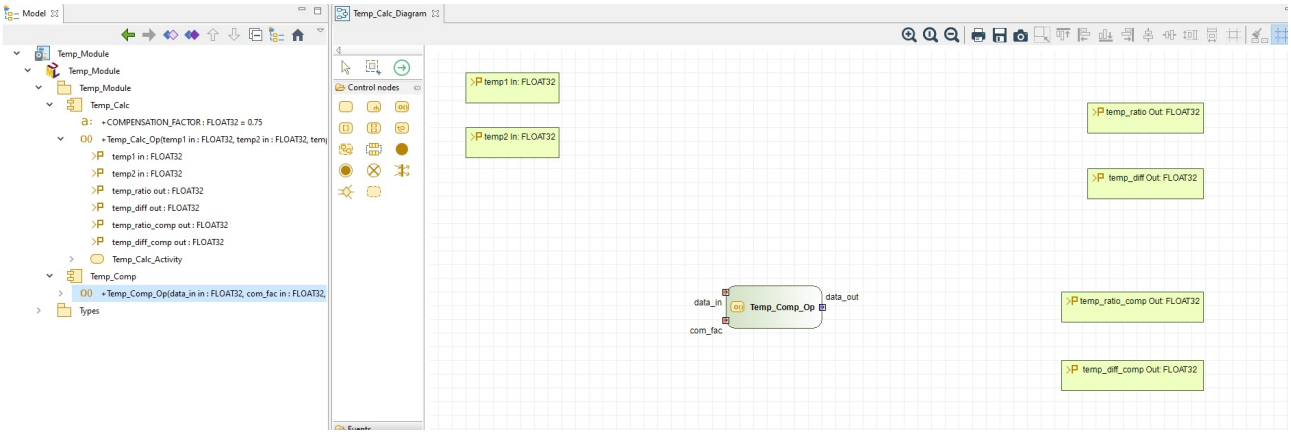
Adding diagram interactions: operations

Operation call is nothing else than invocation of other component operation on activity diagram.

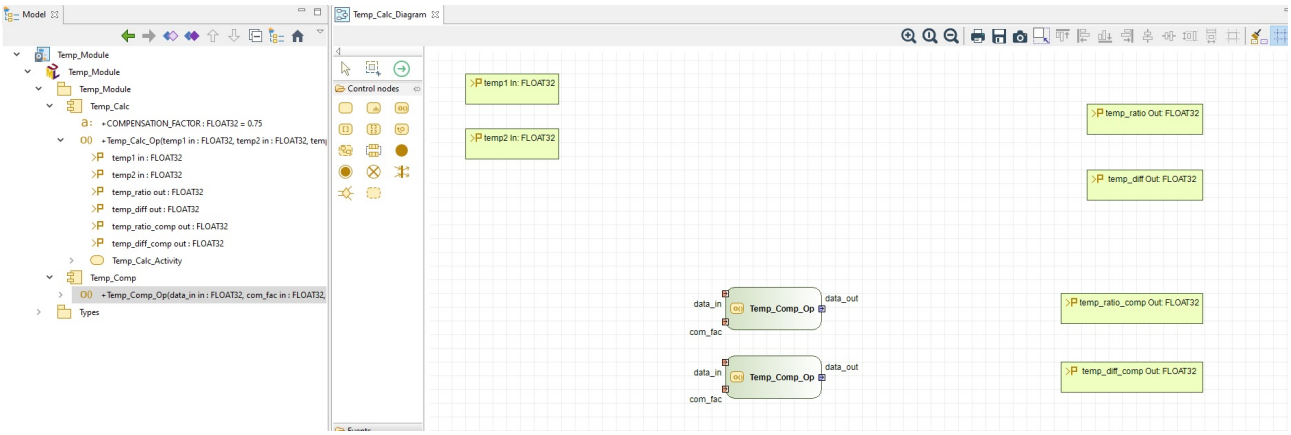
In order to add operation call simply drag and drop other operation into diagram space, then move it to desired position on activity diagram.



Click on each operator pin, then use keyboard arrows to move it to desired position on operation element. Usually input pins should be located on left side of operation element, and output pins should be located on right side of operation element.



Repeat the process to add the rest of required operation calls if needed.



Adding diagram interactions: actions

Action is primitive/build-in interaction type between data elements.

Within Control nodes box select Action – Create an Opaque Action, then click on diagram space where you wish to place new action. Double-click on new action.

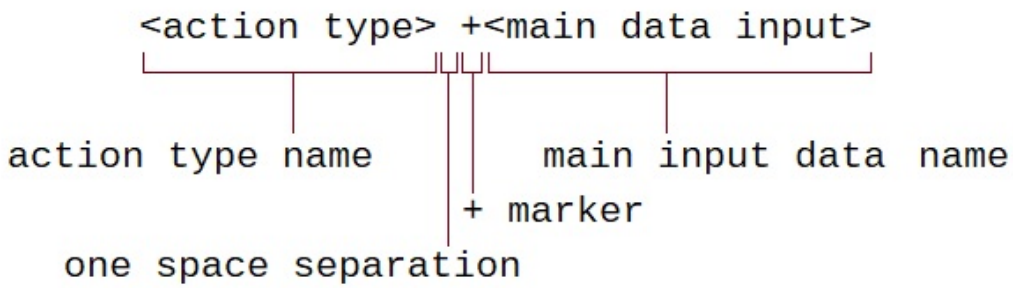
In Name property type desired action type from Table 5.7.1: Action Types.

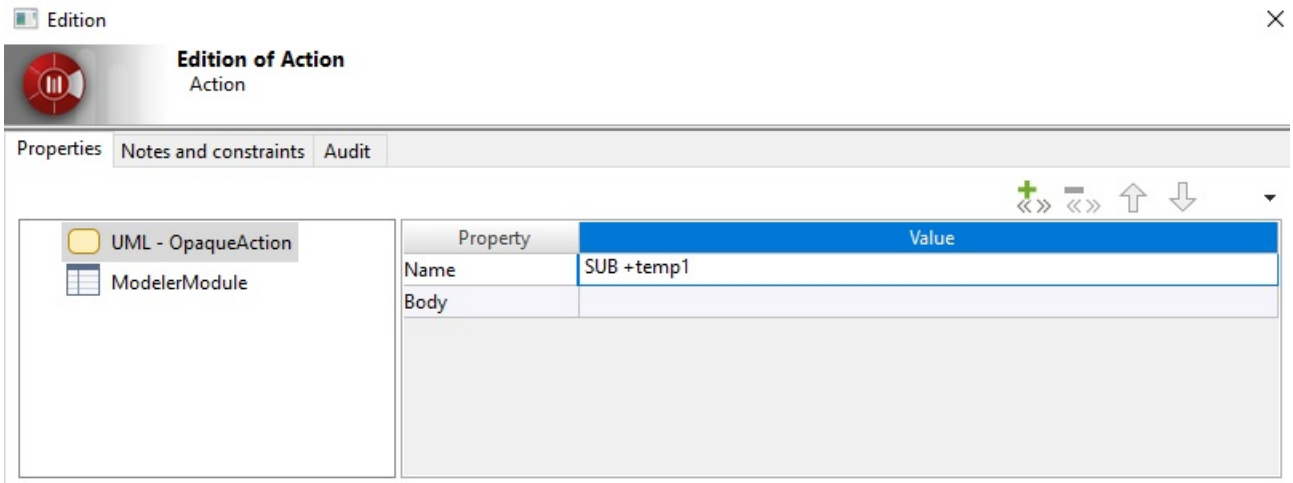
Some kind of actions require to distinguish main data input in order to compute correct results. Please see column **+ *marker required*** in Table 5.7.1: Action Types to find out which actions require to point main data input.

Please consider as example a subtraction between two data elements: *temp1* and *temp2*.

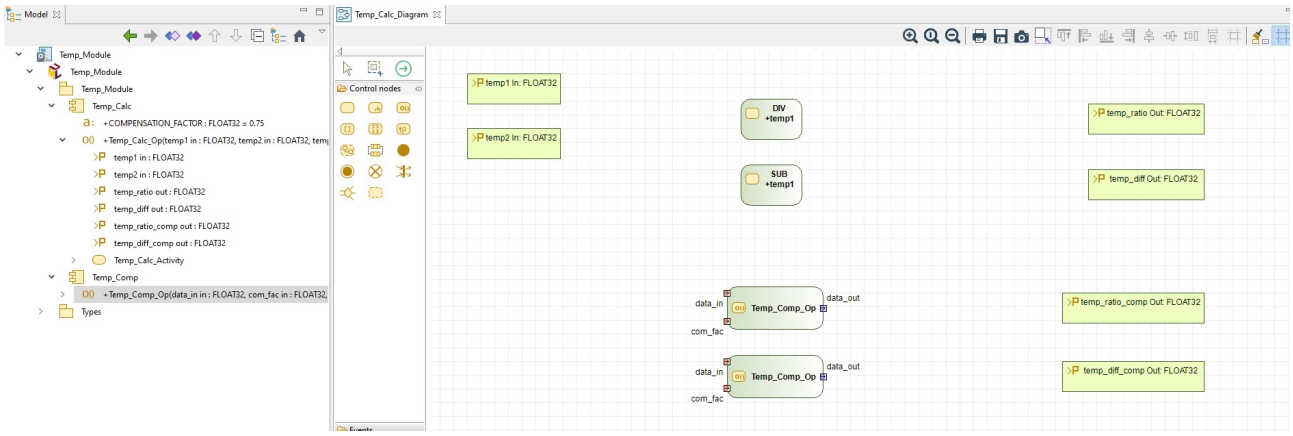
The *temp1-temp2* gives different result than *temp2-temp1*, therefore in such case there is a need to define order by pointing which input data element, *temp1* or *temp2*, should be considered as first one (main) in the equation.

If action type requires to point main data input, then define Name property in the following way:





Repeat the process to add the rest of required action interactions needed.

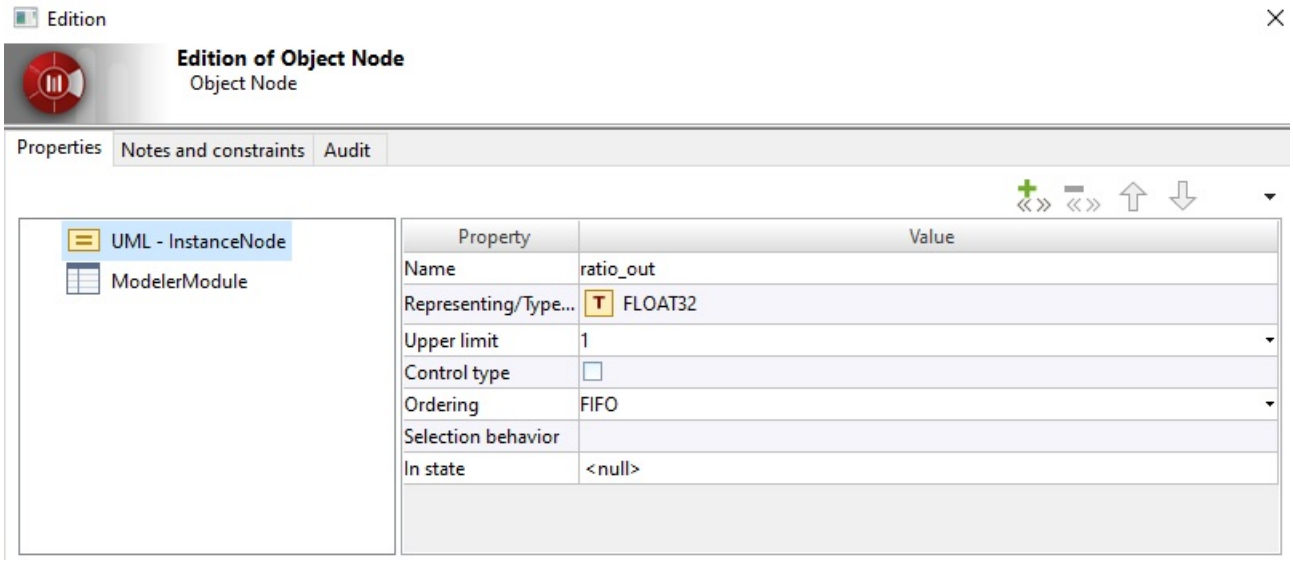


Adding local data elements

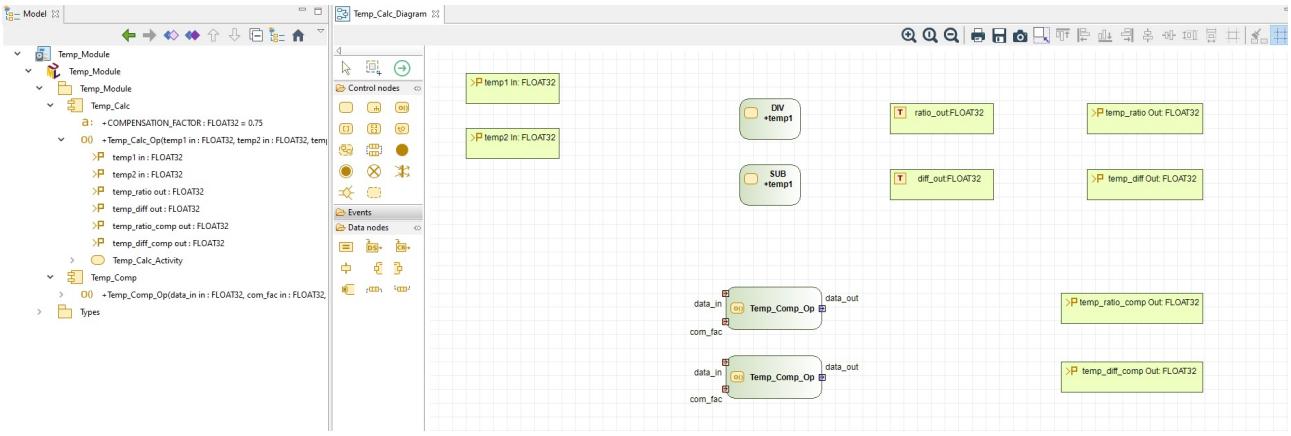
Local data elements can be added to connect interactions, parameters or reuse data in other places of activity diagram.

Within Data nodes box select Object Node – Create an Object Node, then click on diagram space where you wish to place new local data element. Double-click on new data element.

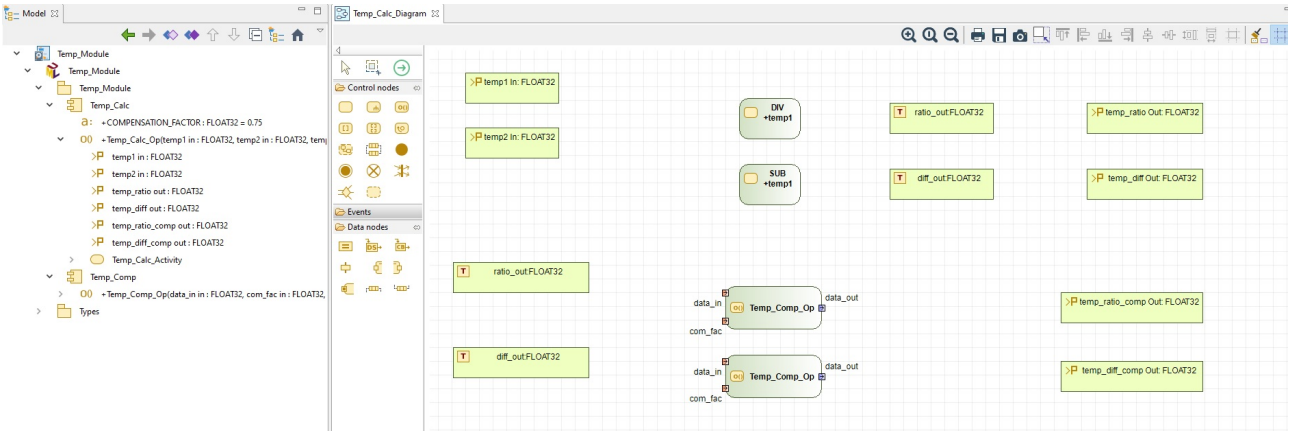
In Name property type desired name of the data element and start typing desired type in Type property, then press “Enter” on keyboard to see auto-completion list (see Create Model Types section for allowed data types).



Repeat the process to add the rest of required local data elements if needed.

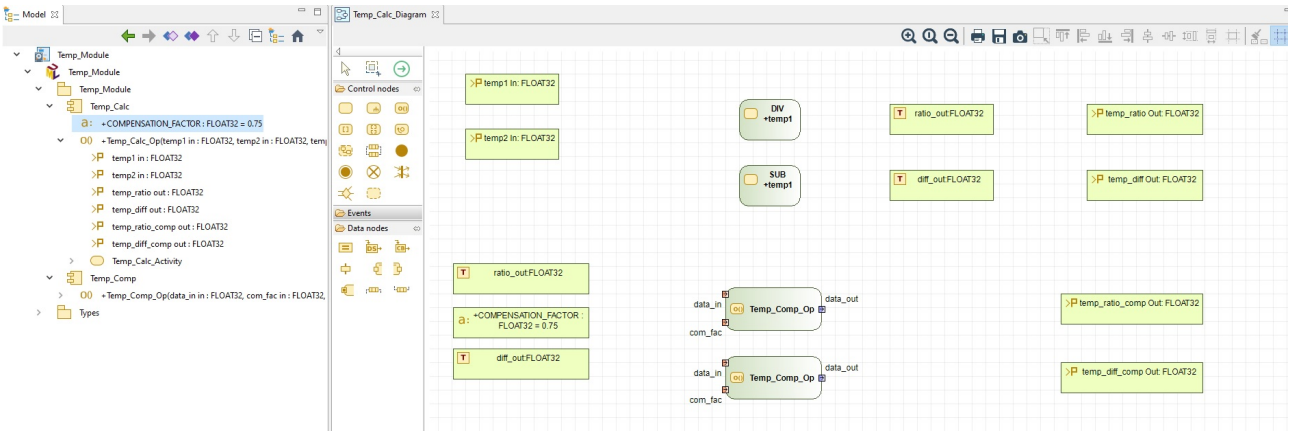


The same data element could be defined again in other place of activity diagram to reuse same data or reduce number of visible connection lines between different elements.



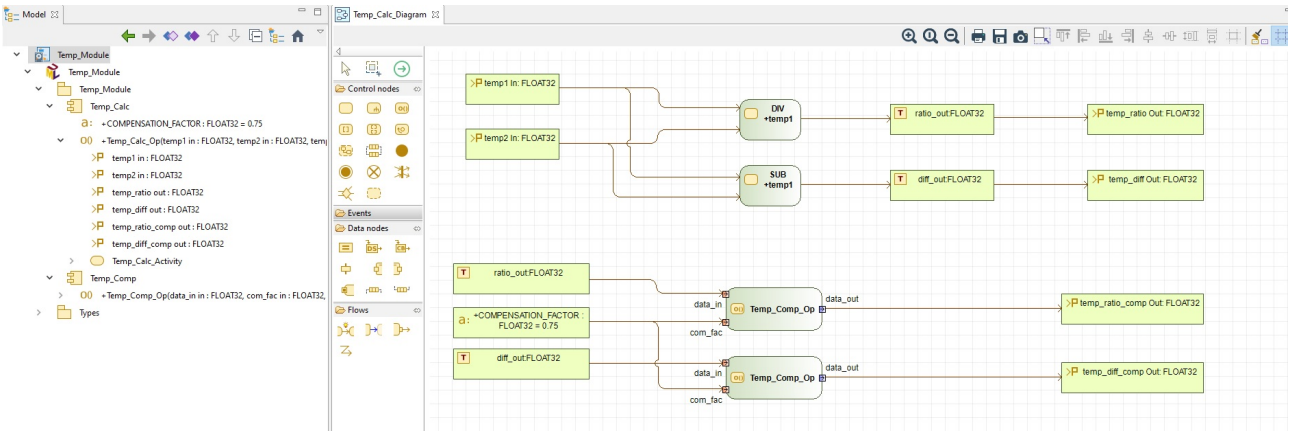
Adding constant data elements

Component constants (previously defined component attributes) can be added to diagram space. In order to add component constant simply drag and drop it into diagram space of any component where it is needed, then move it to desired position on activity diagram.



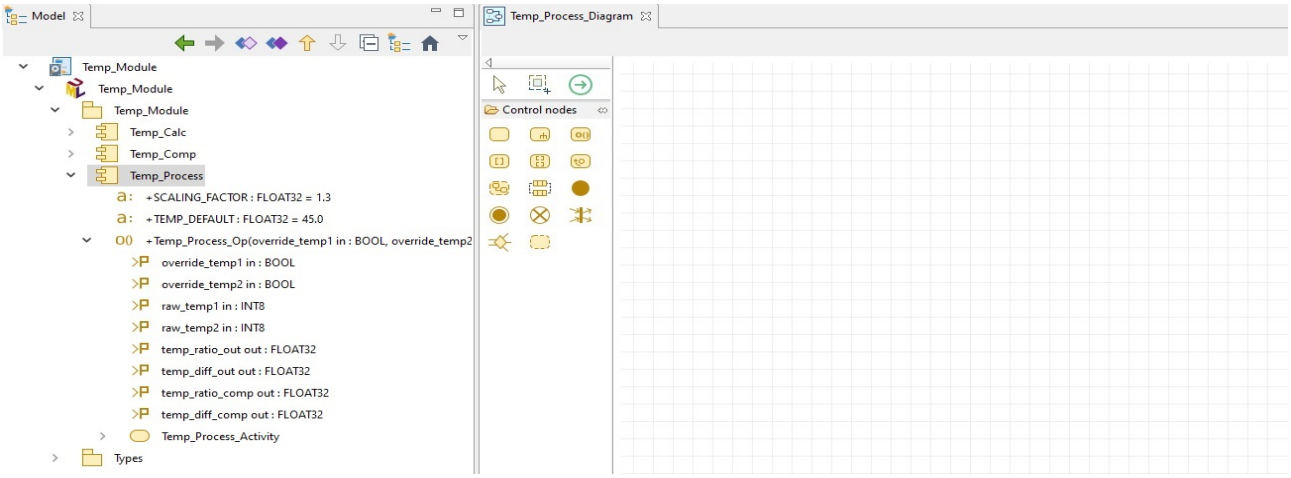
Adding diagram connections

Within Flows box select Object Flow – Create an Object Flow, then connect data and interaction elements on diagram space.

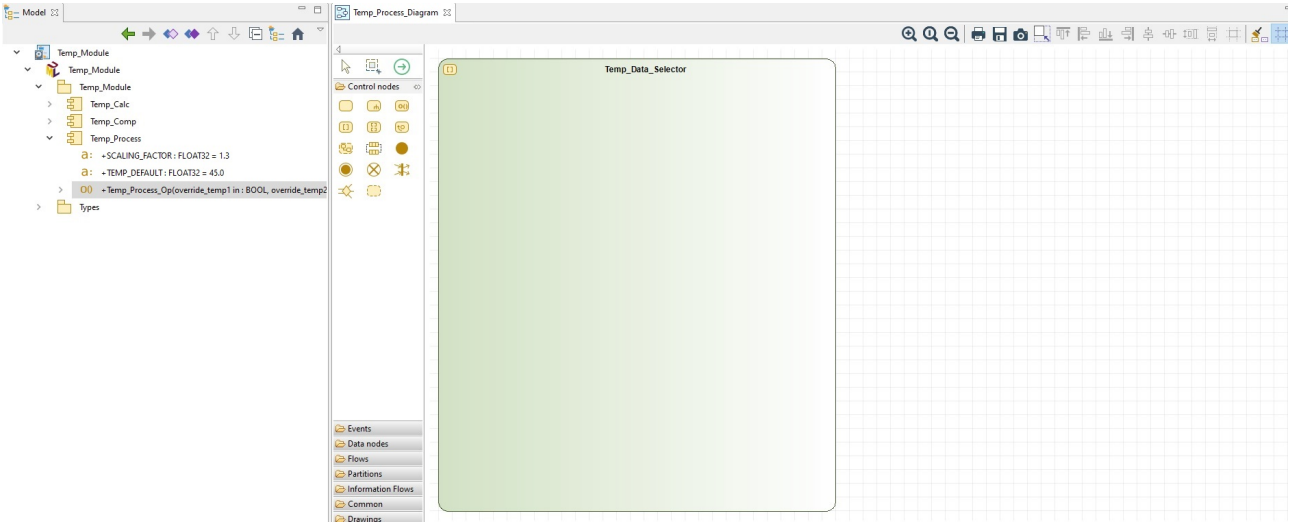


Adding conditional blocks

Conditional blocks can be added to diagram space. Please consider another component with following operation interface, component constants and activity diagram.

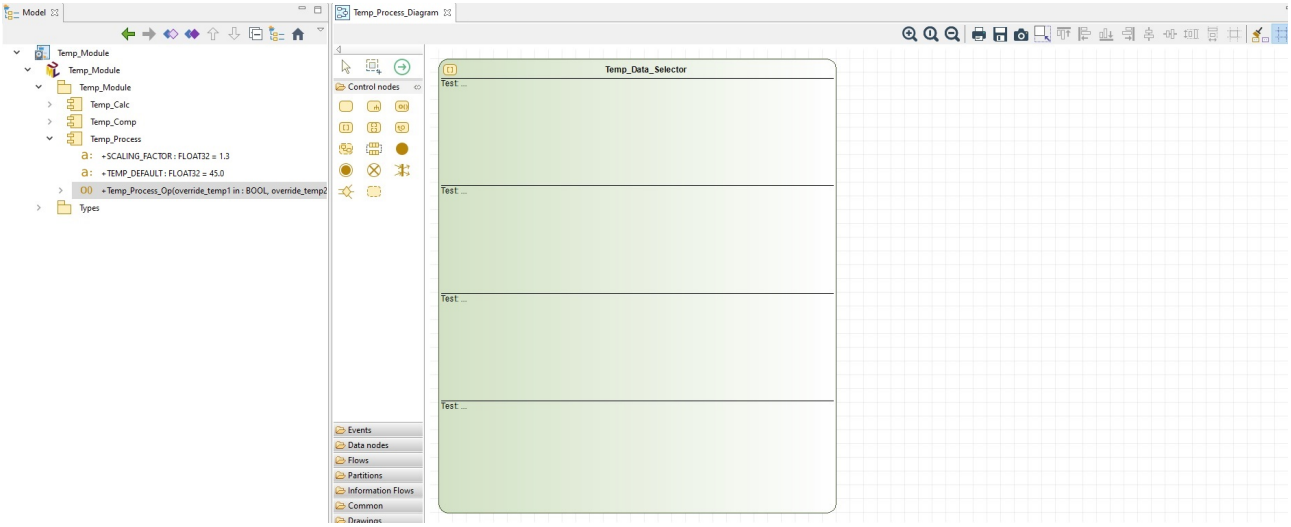


With Control nodes box select Conditional node – Create a Conditional node, then click on diagram space where you wish to place new conditional block. Click on new conditional block, press F2 on keyboard and then give desired name to it.



Conditional branches are represented by clauses. There should be at least two clauses in conditional block, where first clause represents IF conditional branch and last clause represents ELSE conditional branch and any additional branch between first and last is considered as ELSE IF branch.

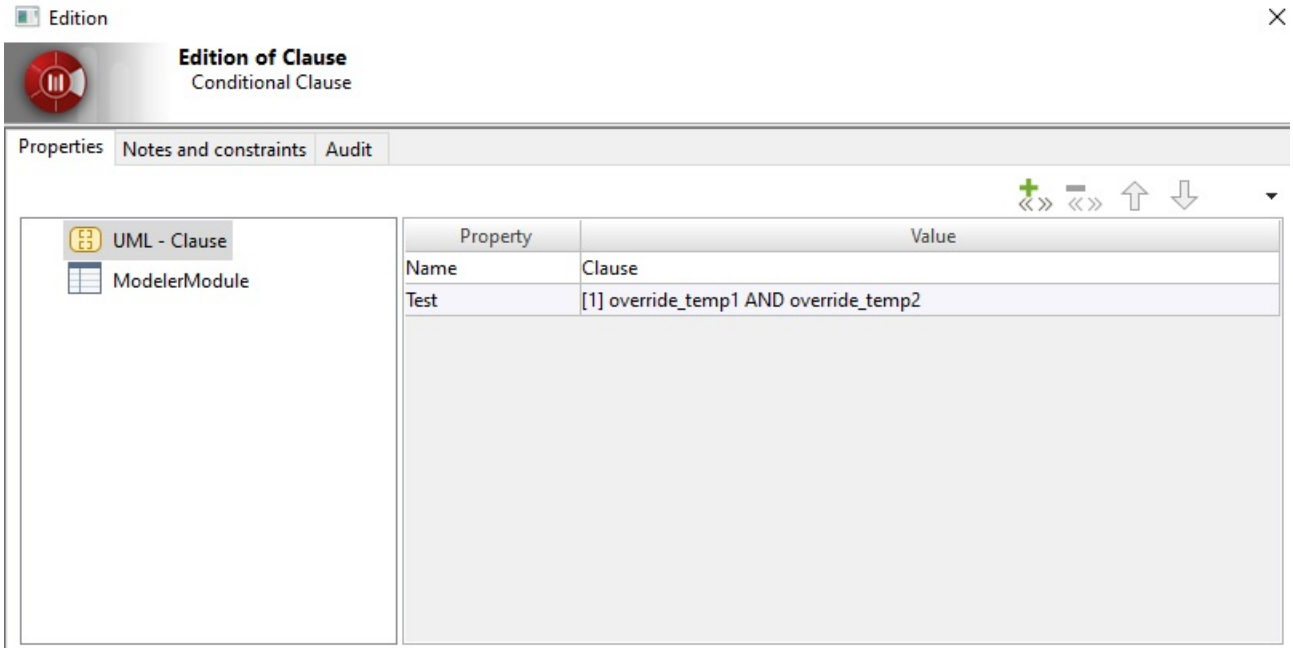
With Control nodes box select Clause – Create a Conditional clause, then click on conditional block to add conditional clause, then repeat the process to add more clauses.



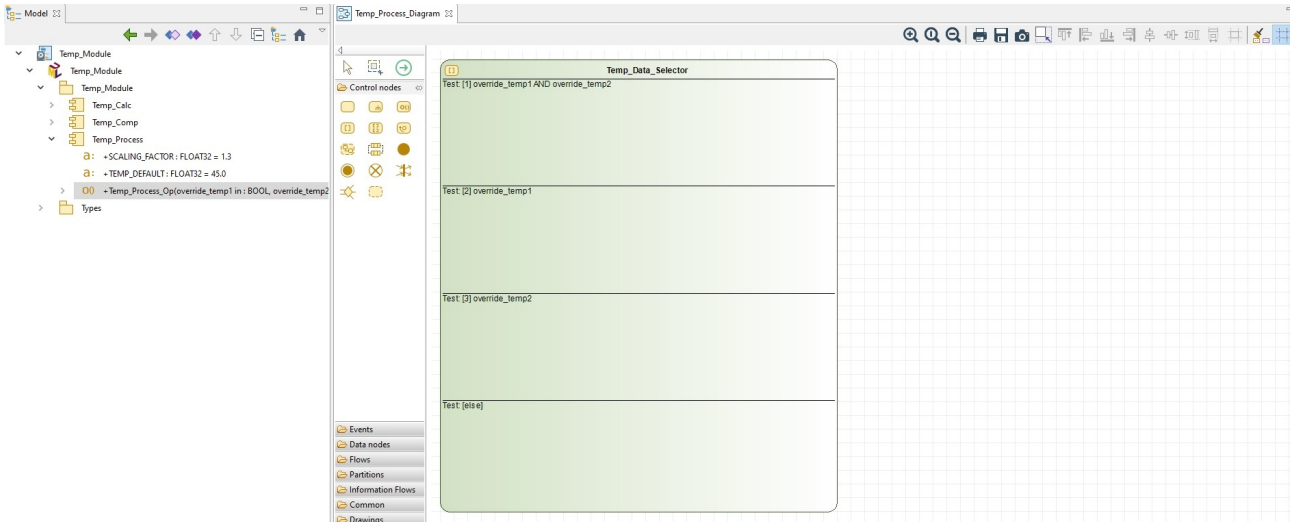
Each clause, apart from last one, requires definition of clause level number and clause decision in Test property of conditional clause as pointed in Modeling Rules.

The clause level numbers are used to define order of conditional branches in entire conditional block. The **[n]** number starts with **[1]** for IF branch and then it is incremented for each next ELSE IF branch that appears in given conditional block.

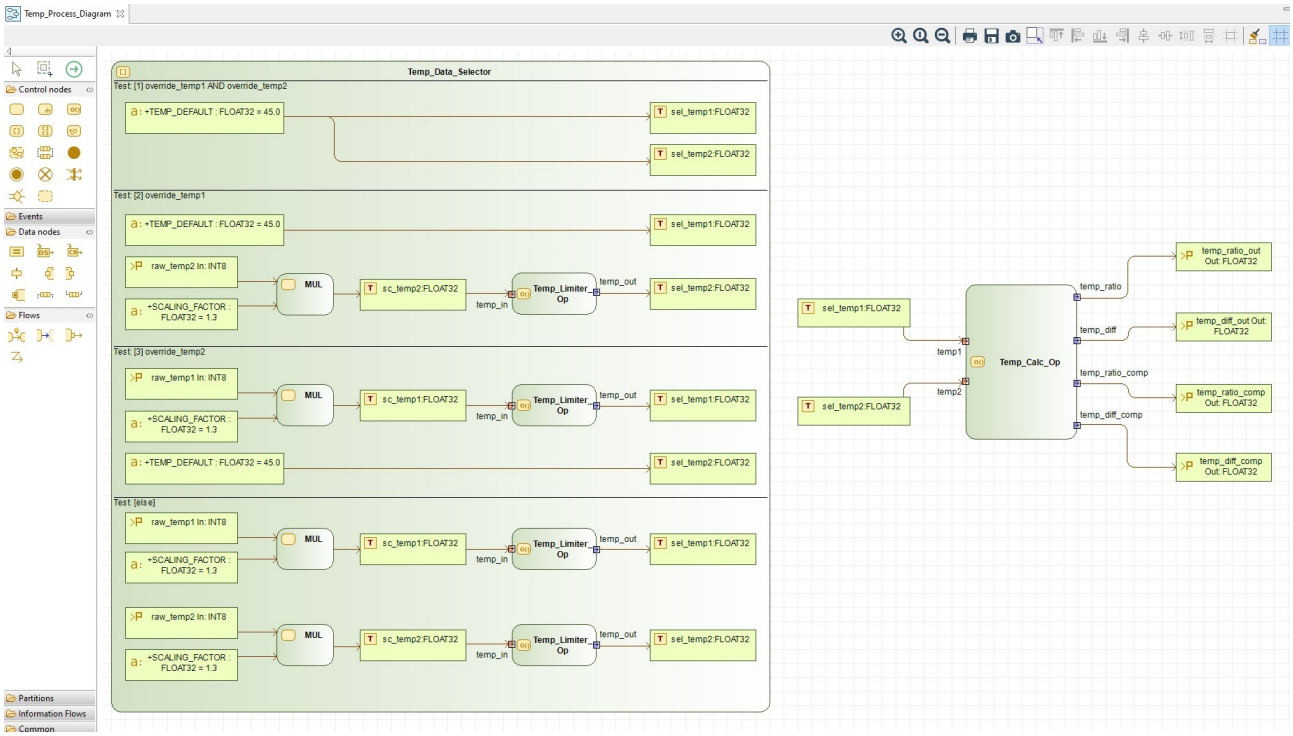
The clause decision is Boolean expression made of input interface parameters/local data elements/constants with (or without) addition of logical/relation type of actions (see Table 5.7.1: Action Types) between them and round brackets **( )** for separation if needed.



The last clause does not have clause level number and Boolean expression. The Test property for last clause must be simply set to **[else]**.



Define component activity elements (data/interactions) within each clause and diagram space.



Create Next Components

To add other components under the same package follow all steps of Model Component Creation section.

5.4 Model Package Creation

Create New Package

Right-click on UML sub-project element and select Create element → Package. Click on new package, press F2 on keyboard and then give desired name to it.

To add other components under this new package follow all steps of Model Component Creation section.

5.5 Modeling Rules

Model Rules

MR#1 RULE

Each model **shall** have following generic layout:

1. <model package 1>
   - <model component 1.1>
   - <model component 1.2>
   - ...
   - <model component 1.n>
2. <mode package 2>
   - <model component 2.1>

    - <model component 2.2>
    - ...
    - <model component 2.n>
3. ...
4. <model package n>
5. <type package>

MR#2 RULE
Each model **shall** have at least one <model package>.

MR#3 RULE
Each <model package> **shall** have at least one <model component>.

MR#4 RULE
Each model **shall** have one <type package>.

MR#5 RULE
<type package> **shall** be named Types.

MR#6 RULE
<type package> **shall** have following types:

- INT8
- INT16
- INT32
- INT64
- UINT8
- UINT16
- UINT32
- UINT64
- FLOAT32
- FLOAT64
- BOOL

Component Attribute Rules

CAR#1 RULE
Each <model component> **can** have one or more <component attributes>.

CAR#2 RULE
Each <component attribute> **shall** use only data type defined under <type package>.

CAR#3 RULE
Each <component attribute> **shall** have different name than all <operation parameters> and <object nodes> under any <component operation>.

CAR#4 RULE
Each <component attribute> **shall** have defined value.

CAR#5 RULE
Each <component attribute> **shall** have at least one output connection on <activity diagram> heading to:

- output <operation parameter>, or
- input of <object node>, or
- input pin of <operation interaction>, or
- input of <action interaction>.

Component Operation Rules

COR#1 RULE
Each <model component> **shall** have one <component operation>.

COR#2 RULE
Each <component operation> **shall** have at least one input <operation parameter>.

COR#3 RULE
Each <component operation> **shall** have at least one output <operation parameter>.

COR#4 RULE
Each <operation parameter> **shall** use only data type defined under <type package>.

COR#5 RULE
Each <operation parameter> **shall** have different name than other <operation parameters> under the same <component operation>.

COR#6 RULE
Each input <operation parameter> **shall** have at least one output connection on <activity diagram> heading to:

- output <operation parameter>, or
- input of <object node>, or
- input pin of <operation interaction>, or
- input of <action interaction>.

COR#7 RULE
Each output <operation parameter> **shall** have one input connection on <activity diagram> coming from:

- <component attribute>, or
- input <operation parameter>, or
- output of <object node>, or
- output pin of <operation interaction>, or
- output of <action interaction>.

Activity Diagram Rules

ADR#1 RULE

Each <component operation> **shall** have one <activity diagram>.

Object Node Rules

ONR#1 RULE
Each <activity diagram> **can** have one or more <object nodes>.

ONR#2 RULE
Each <object node> **shall** use only data type defined under <type package>.

ONR#3 RULE
Each <object node> **shall** have different name than <operation parameters> under the same <component operation>.

ONR#4 RULE
Each <object node> **shall** have one input connection on <activity diagram> coming from:

- <component attribute>, or
- input <operation parameter>, or
- output of other <object node>, or
- output pin of <operation interaction>, or
- output of <action interaction>.

ONR#5 RULE
Each <object node> **shall** have at least one output connection on <activity diagram> heading to:

- output <operation parameter>, or
- input of other <object node>, or
- input pin of <operation interaction>, or
- input of <action interaction>.

Operation Interaction Rules

OIR#1 RULE
Each <activity diagram> **can** have one or more <operation interactions>.

OIR#2 RULE
Each input pin of <operation interaction> **shall** have one input connection on <activity diagram> coming from:

- <component attribute>, or
- input <operation parameter>, or
- output of <object node>.

OIR#3 RULE
Each output pin of <operation interaction> **shall** have one output connection on <activity diagram> heading to:

- output <operation parameter>, or
- input of <object node>.

Action Interaction Rules

AIR#1 RULE
Each <activity diagram> **can** have one or more <action interactions>.

AIR#2 RULE
Each <action interaction> **shall** use only action type defined under Table 5.7.1: Action Types.

AIR#3 RULE
Each <action interaction> **shall** have required number of input and output connections as per Table 5.7.1: Action Types.

AIR#4 RULE
Each <action interaction> **shall** have input connection on <activity diagram> coming from:

- <component attribute>, or
- input <operation parameter>, or
- output of <object node>.

AIR#5 RULE
Each <action interaction> **shall** have output connection on <activity diagram> heading to:

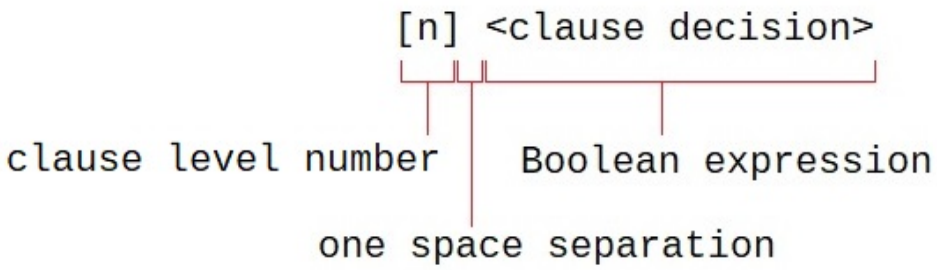
- output <operation parameter>, or
- input of <object node>.

Conditional Interaction Rules

CIR#1 RULE
Each <activity diagram> **can** have one or more <conditional interactions>.

CIR#2 RULE
Each <conditional interaction> **shall** have at least two <conditional clauses>.

CIR#3 RULE
Each <conditional clause> (excluding the last one) **shall** have following definition of clause level number and clause decision:



CIR#4 RULE
[n] clause level number **shall** start with [1] for first <conditional clause> and **shall** be incremented for each next <conditional clause>.

CIR#5 RULE
<clause decision> is Boolean expression that **can** consist:

- input <operation parameter>,
- <object node>,
- <component attribute>,
- <action interaction> of logical/relational type,
- round brackets ().

CIR#6 RULE
Last <conditional clause> **shall** be set to [else].

CIR#7 RULE
Each <conditional clauses> **can** have at lest one or more:

- <operation parameter>, or
- <object node>, or
- <operation interaction>, or
- <action interaction>.

Naming Convention Rules

NCR#1 RULE
Names of model elements **shall** contain only allowed characters listed in Table 5.6.1: Applicable Characters.

NCR#2 RULE
Names of model elements **shall** start with upper or lower case letters only.

NCR#3 RULE
Names of model elements **shall not** contain white spaces.

NCR#4 RULE
Names of model elements **shall** be different than types of <action interactions>.

5.6 Naming Convention

Table 5.6.1: Applicable Characters defines list of applicable characters within name of any model element.

Table 5.6.1: Applicable Characters

Character type	Allowed characters
Upper case letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Lower case letters	a b c d e f g h i j k l m n o p q r s t u v w x y z
Digits	1 2 3 4 5 6 7 8 9 0
Special characters	_ -

5.7 List of Actions

Table 5.7.1: Action Types defines list of possible action interactions between model elements on activity diagram.

Table 5.7.1: Action Types

Action Type	Definition	+ marker required
ADD	Addition arithmetic operation. Requires at least two input connections. Requires exactly one output connection.	
SUB	Subtraction arithmetic operation. Requires at least two input connections. Requires exactly one output connection.	Yes
MUL	Multiplication arithmetic operation. Requires at least two input connections. Requires exactly one output connection.	
DIV	Division arithmetic operation. Requires at least two input connections. Requires exactly one output connection.	Yes
AND	And logical operation. Requires at leas two input connections. Requires exactly one output connection.	
OR	Or logical operation. Requires at leas two input connections. Requires exactly one output connection.	
NOT	Not logical operation. Requires exactly one input connection. Requires exactly one output connection.	
BAND	And bitwise operation. Requires at least two input connections. Requires exactly one output connection.	
BOR	Or bitwise operation. Requires at least two input connections. Requires exactly one output connection.	
BXOR	Xor bitwise operation. Requires at least two input connections. Requires exactly one output connection.	
BNOT	Not bitwise operation. Requires exactly one input connection. Requires exactly one output connection.	

Action Type	Definition	+ marker required
BLS	Bit left shift operation. Requires exactly two input connections. Requires exactly one output connection.	Yes
BRS	Bit right shift operation. Requires exactly two input connections. Requires exactly one output connection.	Yes
EQ	Equal to relation operation. Requires exactly two input connections. Requires exactly one output connection.	
NE	Not equal to relation operation. Requires exactly two input connections. Requires exactly one output connection.	
GT	Greater than relation operation. Requires exactly two input connections. Requires exactly one output connection.	Yes
LT	Less than relation operation. Requires exactly two input connections. Requires exactly one output connection.	Yes
GE	Greater than or equal to relation operation. Requires exactly two input connections. Requires exactly one output connection.	Yes
LE	Less than or equal to relation operation. Requires exactly two input connections. Requires exactly one output connection.	Yes

6 How to Run MCG

Please make sure that installation steps described in Installation sections were completed and that two environment variables, i.e. "MCG_CC" and "MCG_CGC", were created.

6.1 Converter Component (MCG CC)

Run Converter Component

The MCG CC requires two command line arguments to run:

- <model_dir_path> Path to model directory, where all catalogs with .exml files are stored
- <output_dir_path> Path to output directory, where results from MCG CC will be saved

To run the MCG CC please type in a command line window, keeping the specific order of arguments:

```
%MCG_CC% "<model_dir_path>" "<output_dir_path>"
```

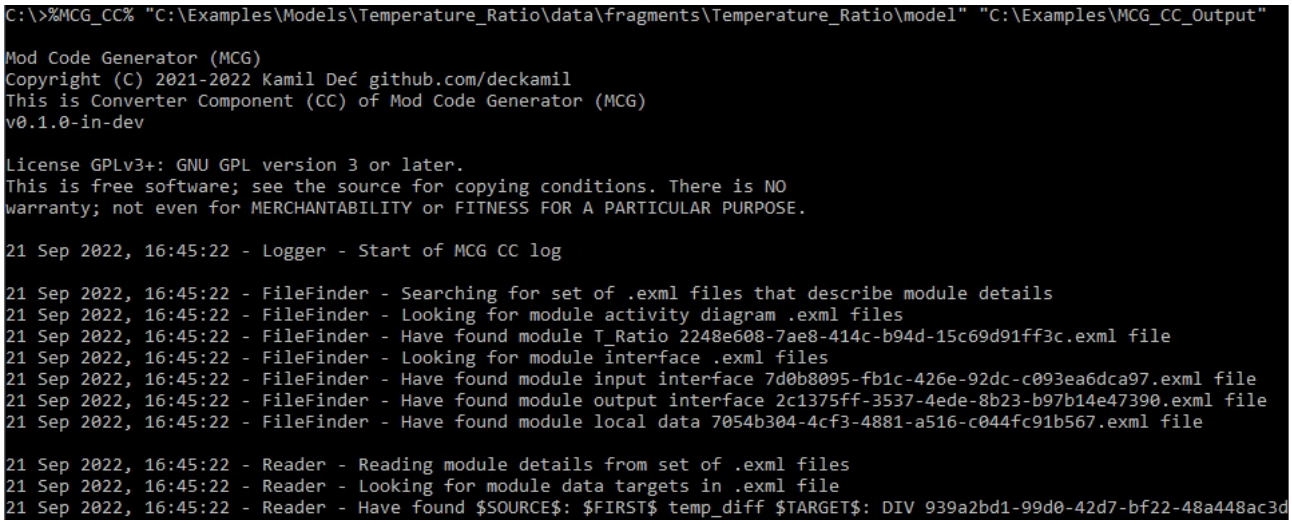
The "<model_dir_path>" is usually equal to "<model_project_path>\data\fragments\<model_name>\model".

Please consider example where:

- the "<model_dir_path>" is equal to "C:\Examples\Models\Temperature_Ratio\data\fragments\Temperature_Ratio\model", and
- the "<output_dir_path>" is equal to "C:\Examples\MCG_CC_Output"

In the above example the MCG CC should be invoked as:

```
%MCG_CC% "C:\Examples\Models\Temperature_Ratio\data\fragments\Temperature_Ratio\model" "C:\Examples\MCG_CC_Output"
```



Check Converter Component Outputs

The MCG CC generates the MCG CGC configuration file and the MCG CC log file. It is recommended to check both to ensure that the MCG CC run was completed successfully.

Check the configuration file

At minimum inspection of the configuration file should check presence of mandatory header and footer, composed from start (for header) or end (for footer) markers, along with their date markers.

The MCG CGC configuration file shall have following header:

MCG CGC CONFIG START <date_marker>

The MCG CGC configuration file shall have following footer:

MCG CGC CONFIG END <date_marker>

No text shall appear after the footer within the configuration file. The <date_marker> for the header is expected to be older than <date_marker> for the footer.

Check the log file

At minimum inspection of the log file should check presence of mandatory header and footer, composed from start (for header) or end (for footer) markers, along with their date markers.

The MCG CC log file shall have following header:

<date_marker> - Logger - Start of MCG CC log

The MCG CC log file shall have following footer:

<date_marker> - Logger - End of MCG CC log

The MCG CC log file shall **not** contain any records from error handler:

<date_marker> - ErrorHandler - ERRORS FOUND, Mod Code Generator (MCG) Converter Component (CC) WILL EXIT
<date_marker> - ErrorHandler - <error_code>: <error_description>

If the error handler record occurs within the MCG CC log file, then it means that error (or more) was detected during conversion of the model into the configuration file format.

No text shall appear after the footer within the log file. The <date_marker> for the header is expected to be older than <date_marker> for the footer.

6.2 Code Generator Component (MCG CGC)

Run Code Generator Component

The MCG CGC requires two command line arguments to run:

- <config_file_path> Path to configuration file, which contains source data to code generation
- <output_dir_path> Path to output directory, where results from MCG CGC will be saved

To run the MCG CGC please type in a command line window, keeping the specific order of arguments:

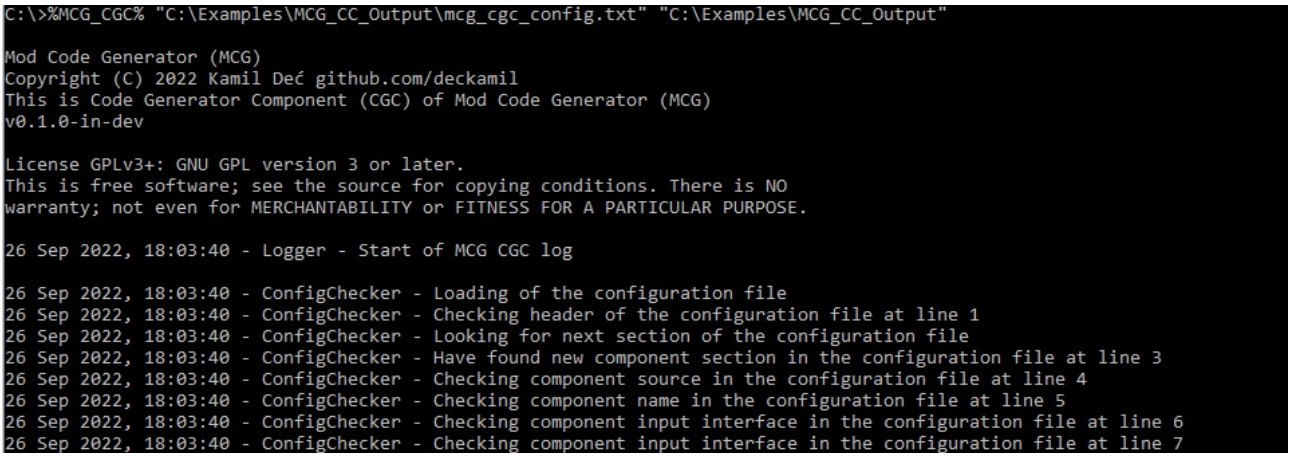
%MCG_CGC% "<config_file_path>" "<output_dir_path>"

Please consider example where:

- the "<config_file_path>" is equal to "C:\Examples\MCG_CC_Output\mcg_cgc_config.txt", and
- the "<output_dir_path>" is equal to "C:\Examples\MCG_CC_Output"

In the above example the MCG CGC should be invoked as:

%MCG_CGC% "C:\Examples\MCG_CC_Output\mcg_cgc_config.txt" "C:\Examples\MCG_CC_Output"



Check Code Generator Component Outputs

The MCG CGC generates source code items and the MCG CGC log file. It is recommended to check each generated source code item and the log file to ensure that the MCG CGC run was completed successfully.

Check the source code item

At minimum inspection of each source code items should check presence of mandatory header and footer.

The source code item shall have following header:

```
/*
*   Generated with Mod Code Generator (MCG) Code Generator Component (CGC)
*   on <date_marker>
*
*   This is source file of <module_name> module.
*/
```

The source code item shall have following footer:

```
/*
* END OF MODULE
*/
```

No text shall appear after the footer within the source code file. The <date_marker> in the header is expected to be the same across each generated source code item. The header is expected to have as well reference to model <module_name> along with its <exml_id_number> from which the source code item was generated.

Check the log file

At minimum inspection of the log file should check presence of mandatory header and footer, composed from start (for header) or end (for footer) markers, along with their date markers.

The MCG CGC log file shall have following header:

<date_marker> - Logger - Start of MCG CGC log

The MCG CGC log file shall have following footer:

<date_marker> - Logger - End of MCG CGC log

The MCG CGC log file shall **not** contain any records from error handler:

> *<date_marker> - ErrorHandler - ERRORS FOUND, Mod Code Generator (MCG) Code Generator Component (CGC) WILL EXIT*
> *<date_marker> - ErrorHandler - <error_code>: <error_description>*

If the error handler record occurs within the MCG CGC log file, then it means that error (or more) was detected during conversion of the configuration file into source code items.

No text shall appear after the footer within the log file. The <date_marker> for the header is expected to be older than <date_marker> for the footer.

7 Configuration File Syntax and Format

This section is not defined yet in current release.