# 1. Signal Flow 2 Flat Signal Flow Example

This example converts models of the SignalFlow paradigm (a hierarchical data flow paradigm) to the equivalent models in the FlatSF paradigm (a flat data flow representation with buffers on the edges)

## 1.1. Directory Organization

- 📁 **SF2FSF**
  - SignalFlow2FlatSF.mga - SignalFlow 2 FlatSF transformation
  - SignalFlow2FlatSF.xme - Transformation exported XML
  - SignalFlow2FlatSF_test_globalObject.mga - Global namespace example
  - SignalFlow2FlatSF_test_globalObject.xme - Exported XML of above
  - SignalFlow2FlatSF_test_sortingFunc.mga - Sorting function example
  - SignalFlow2FlatSF_test_sortingFunc.xme - Exported XML of above
  - SignalFlow_test_distinguished.mga - Distinguished cross product example
  - SignalFlow_test_distinguished.xme - Exported XML of above
  - 📁 **Meta**
    - 📁 **Icons** - Icons for the SignalFlow and FlatSF paradigms
    - SignalFlow.mga - SignalFlow metamodel
    - SignalFlow.xme - SignalFlow.mga exported to XML
    - GS_SignalFlow.xmp - SignalFlow paradigm file
    - FlatSF.mga - FlatSF metamodel
    - FlatSF.xme - FlatSF.mga exported to XML
    - GS_FlatSF.xmp - FlatSF paradigm file
  - 📁 **Models**
    - SignalFlow_1.mga - Model of SignalFlow paradigm
    - SignalFlow_2.mga - Model of SignalFlow paradigm
    - SignalFlow_3.mga - Model of SignalFlow paradigm
    - SignalFlow_4.mga - Model of SignalFlow paradigm
    - SignalFlow_5.mga - Model of SignalFlow paradigm
    - SignalFlow_6.mga - Model of SignalFlow paradigm
    - SignalFlow_7.mga - Model of SignalFlow paradigm
    - SignalFlow_to_connect.,mga - Model of SignalFlow paradigm
    - SignalFlow_1.xme - SignalFlow_1.mga exported XML
    - SignalFlow_2.xme - SignalFlow_2.mga exported XML
    - SignalFlow_3.xme - SignalFlow_2.mga exported XML
    - SignalFlow_4.xme - SignalFlow_2.mga exported XML
    - SignalFlow_5.xme - SignalFlow_2.mga exported XML
    - SignalFlow_6.xme - SignalFlow_2.mga exported XML
    - SignalFlow_7.xme - SignalFlow_2.mga exported XML
    - SignalFlow_to_connect.xme - SignalFlow_2.mga exported XML
  - 📁 **Gen** - Will contain the CG tool generated files
  - 📁 **Udm** - Will contain the Udm meta files

## 1.2. How to run SignalFlow 2 FlatSF example?

Open SignalFlow 2 FlatSF transformation model

- Directly open $/ SignalFlow2FlatSF.mga, if it fails, open GME, choose File/Import XML, and choose $/ SignalFlow2FlatSF.xme

SignalFlow2FlatSF.mga contains the transformation rules, UDM compatible meta information paradigms and configuration information. Following is the folder structure which is shown in browser:

- SignalFlow2FlatSF
  - CrossLinks - UML class diagram for cross reference associations
  - FlatSF - FlatSF Metamodel in UML class diagram format
  - SignalFlow - SignalFlow Metamodel in UML class diagram format
  - zt_SF2FSF - Folder containing the transformations
  - zz_Config - Folder containing configuration information

Run the SignalFlow 2 FlatSF transformation model

- Invoke the GReAT Master Interpreter with icon (**This is a required step for the first time running**), Use the default file paths and names provided.
- The transformations can be invoked in various ways
  1. GR Engine – Performs the transformations in an interpretive manner
  2. GR Debugger – Provides a user interface and debugging features such as break points, single step, step into etc.
  3. Code generator – Converts the transformation into code that can be compiled and executed.
- To run GR Engine, it could be done either :
  - In the GReAT Master Interpreter dialog, check the box "Run GR Engine"; **-or-**
  - Directly invoke the GR Engine interpreter with icon .
  - The default input file is $/Models/SignalFlow_1.mga
  - The output files will be $/Models/outSF1.mga
- To run the GR Debugger
  - Open a command prompt and go to the sample directory $/. Invoke GRD by calling GRD.exe , then load the config file $/config.mga **-or-**
  - Directly invoke the GR Debugger interpreter with icon .
- To run Code Generator, it could be done either :
  - In the same dialog of GReAT Master Interpreter, check the box of "Run Code Generator"; **-or-**
  - Directly invoke the Code Generator interpreter with icon ; **-or-**
  - Open a command prompt and go to the sample directory $/. Invoke CG by calling CG.exe , with config file $/config.mga
  - After the files have been generated open the generated Visual Studio project using VS71 and compile the project
  - You can run the generated code with default arguments by setting the working directory to be ..\ and Program argument to be –d (default)

# 2. The Global Object, Sorting and Distinguished Merging Examples

You can follow the same steps listed above to run these examples. The UMT files for these examples are as below:

1. SignalFlow2FlatSF_test_globalObject.xme – This is the example demonstrates the usage of Global Objects in GReAT. Observe that first a RootContainer is created in rule RootRule, which is associated with global root object TempRoot. Then, in a non-adjacent subsequent rule, called CreateQueue, RootRule is found starting from TempRoot, and neither object is supplied by input ports.
2. SignalFlow2FlatSF_test_sortingFunc.xme – This example demonstrates sorting in GReAT. In rule PortBase, PortBase objects are passed along to subsequent rules in a specific order determined by the compare function nameCmpFunc (attribute of output port LOut). The compare function nameCmpFunc is defined in zz_Config\CodeLibrary. Rule CreateQueue will create Queue objects in this order (check the output file Models\OutFSF.xml)
3. SignalFlow_test_distinguished.xme – This example demonstrates distinguished merging in GReAT. Distinguished merging is used in two rules in this example: rule SelectComponents, and rule SelectPorts (see the distinguished cross product attribute of these rules.) SelectComponents selects component pairs that are laid out horizontally right next to each other. The pattern matching first finds all possible ordered pairs from the set of input components. To ensure that component ”From” lies to the left of component ”To”, we utilize the compare function ”XPosCmp” in a guard condition ”From is left to To”, which discards all pairs where ”From” is right to ”To”. Finally, the distinguished merging gets rid of all non-adjacent pairs. These source-destination component pairs are then passed along one-by-one to the rule SelectPorts. SelectPorts selects “OutputPort”-s and ”InputPort”-s in ”BaseComponent”-s. Here, the same predicate “YPosCmp” used as a compare function for both output ports. The selected component ports are connected in a subsequent rule, called ConnectPorts.

Please refer to the GReAT user manual for more information on these features.