# What is it?

ModellMeister gives you the ability to create a dataflow-driven, block-oriented system. By combining blocks and their containing algorithms with wires by a simple text file, you can reconfigure the system in a very fast way.

ModellMeister transforms this text file into static C# code which does not have any reflection overhead and can be executed directly.

# Tutorial

## Download

ModellMeister is available at GitHub and can be downloaded via:

<https://github.com/mbrenn/modellmeister/releases>

You need to extract the ZIP-Archive in whatever directory, you like to.

## The Prototype tool

By executing the binary “mbgi\_gui.exe” via double-click in Exporer, the prototype tool opens with an empty workspace.

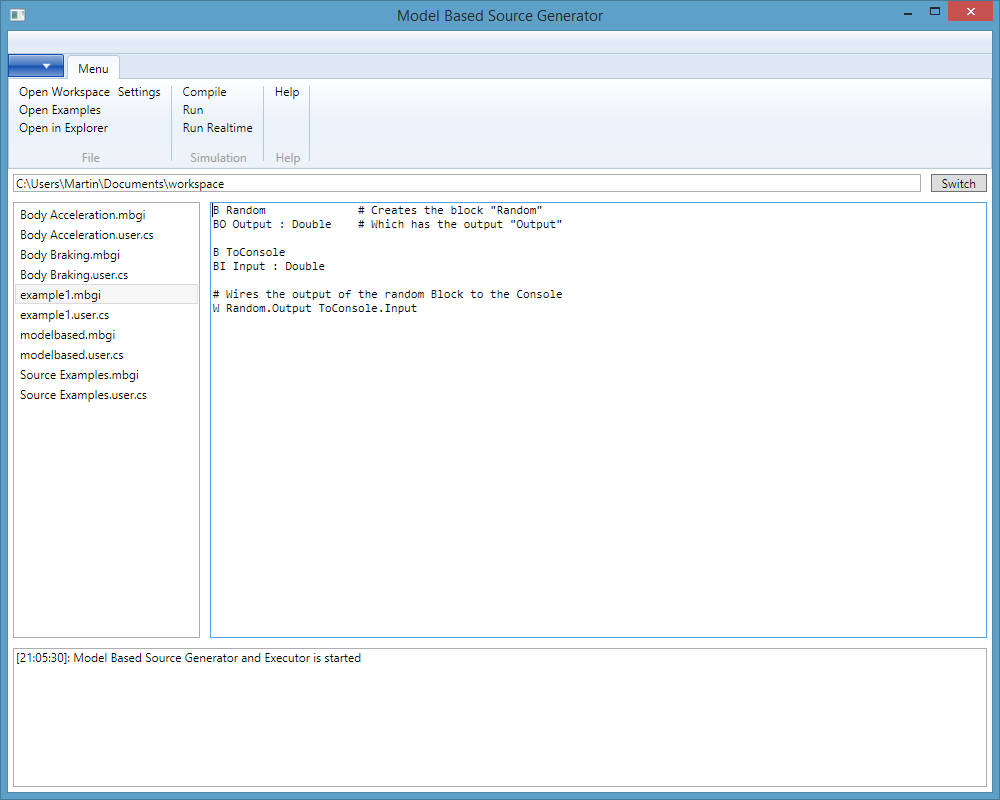


Image 1 - Model Based Source Generator

A new folder “workspace” in your Document-Directory will be created.

To get a first glance about the tool and its capabilities, you can load one of the examples by clicking on “Open Examples” and selecting one of the .mbgi files. To run the simulation simulation, click on “Run”.

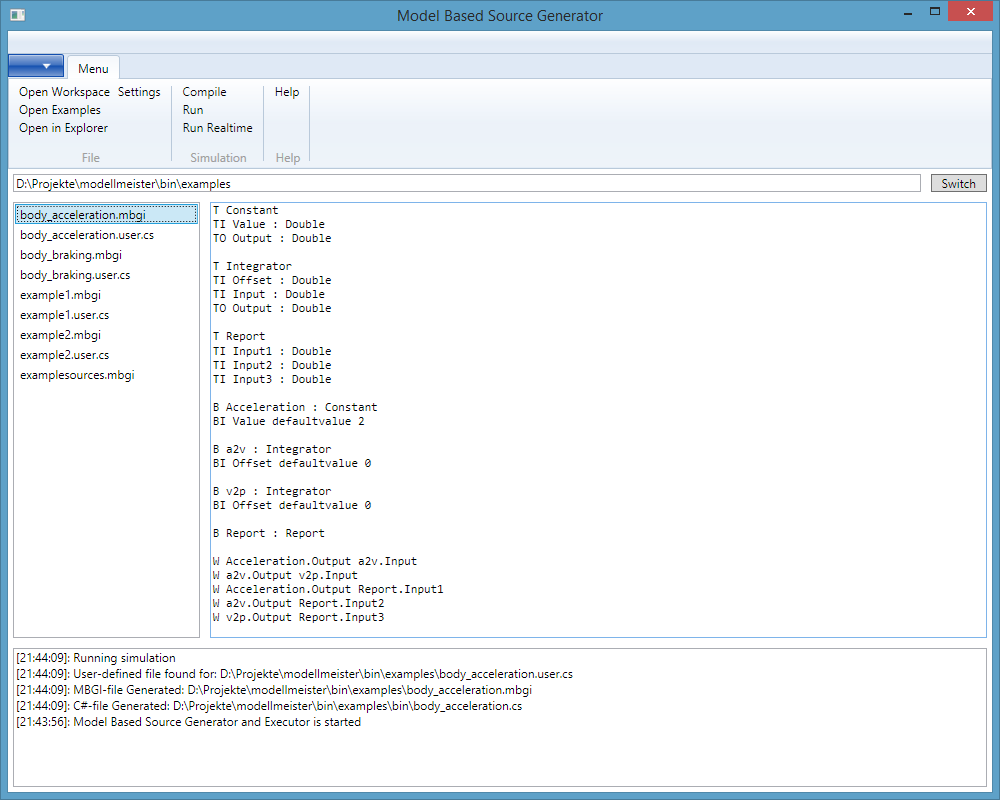
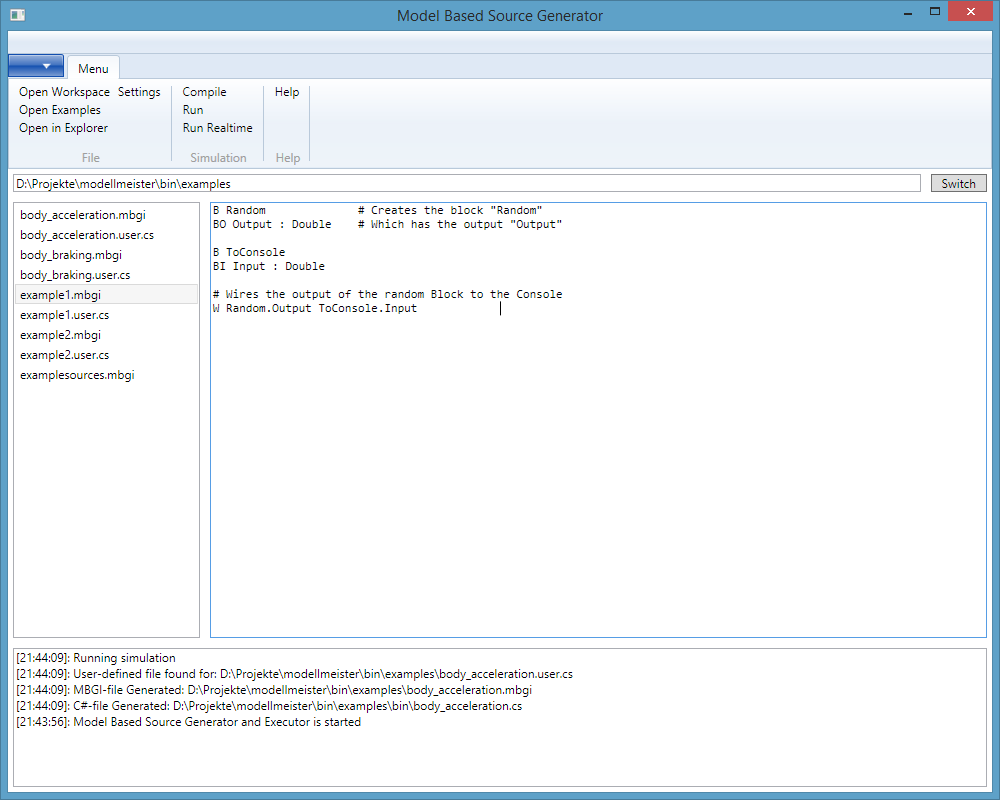


Image 2 – Body Acceleration.mbgi

## Tutorial – Example 1

After the loading of an example, the MBGI-Source and C# Source-Code is shown in the window.



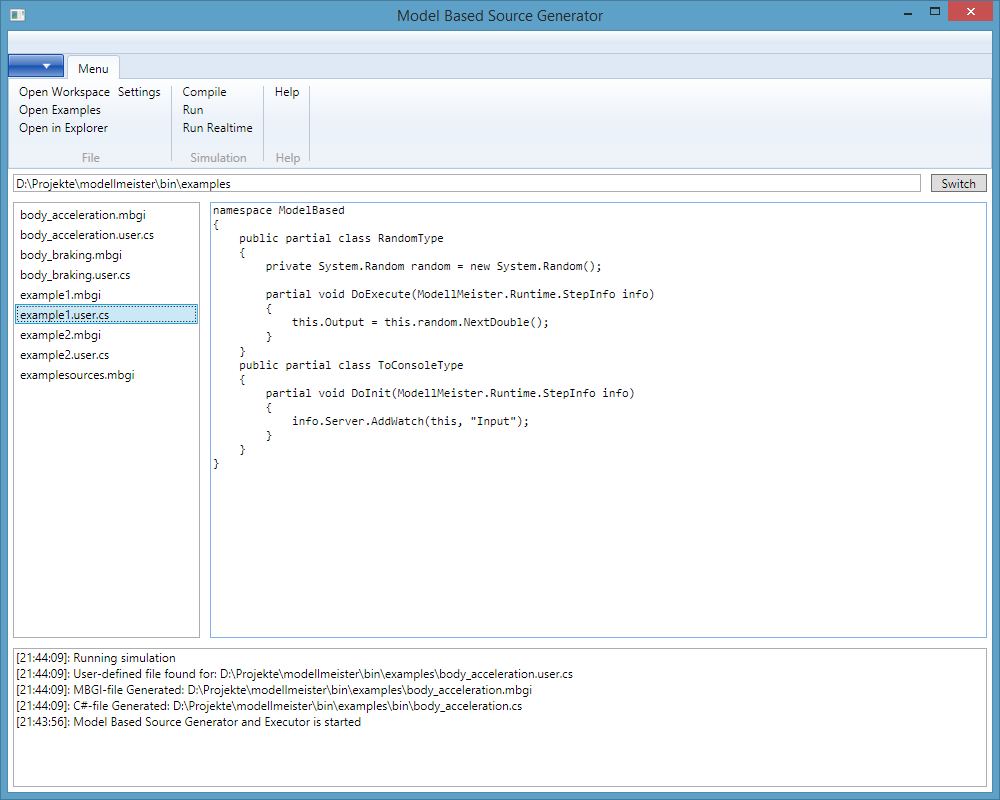


Image 3 - Example Project

The first example describes the following architecture:

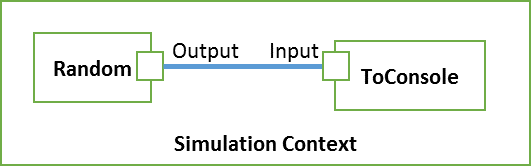


Image 4 - Model for example 1

The MBGI Source on the left side defines the model-elements and their connections to each other. The C#-Code of the right side defines the input-output behavior of the model-element. It is also called implementation.

The Random block is defined by the following two statements:

B Random # Creates an instance of the block "Random"

BO Output : Double # Which has the output "Output"

As defined in the comment, a new instance of an untyped block is created. The corresponding C#-Type will be auto-generated and will get the name of the instance with the postfix “Type”.

The implementation of the block is done on the right side of the tool:

public partial class RandomType  
 {  
 private System.Random random = new System.Random();

partial void DoExecute(ModellMeister.Runtime.StepInfo info)  
 {   
 this.Output = this.random.NextDouble();  
 }  
 }

Whenever the output of the block is needed, the function “DoExecute” will be called by the framework. The implementation creates a new random number and stores it into the Port “this.Output”.

The block “ToConsole” is created in a similar way. Whatever value will be received, it stores the value into the Debug-Interface, so it can be used by the simulation framework.

The two blocks are connected via the wire statement:

# Wires the output of the random Block to the Console

W Random.Output ToConsole.Input

It takes the output of the “Random” instance and wires it to the input of the block “ToConsole”. Before the execution method of the block “ToConsole” is called, the information is transferred from the “Random” block to the input. This is task of the framework and the autogenerated code.

You can start the simulation by clicking “Run Simulation” and you will get a diagram, where all the random values are printed.



Image 5 - Diagram with Random values

The framework performs the following steps:

* Generation of the C#-Code out of the .mbgi file
* Compilation of the two C#-Files to a binary .Net-library (.dll)
  + The autogenerated one
  + The user-defined one in the tool
* The library (.dll) is loaded and the blocks are called by the simulation framework.

To view the autogenerated file, you can open the workspace by clicking “Open workspace”.

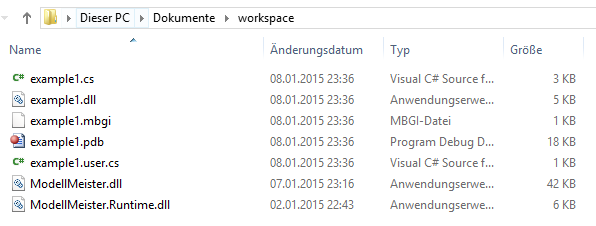


Image 6 - Workspace

# Entities

An example for a model is given in the following diagram:

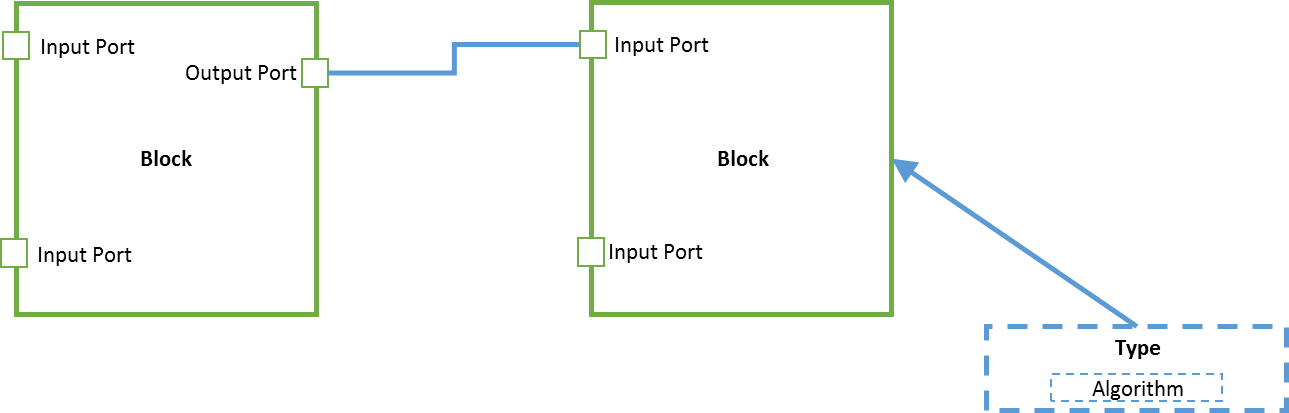


Image 7 - Entity Types

**Block** – A block is an instance of a certain block type. It has inputs and outputs and the algorithm which is defined in the associated block type.

**Port** – Every input and output is also a port. A port can be whether an input or an output. A port is always associated to a block or a type.

**Wire** – A wire connects the output of a block, which is an instance of a type, to an input of another block.

**Type** – A type is a template for the blocks and defines the inputs, the outputs and a corresponding algorithm. The algorithm cannot be executed since it is not instantiated.

**Algorithm** – The algorithm is implemented by the user and performs a calculation of outputs by the inputs.

# Block types

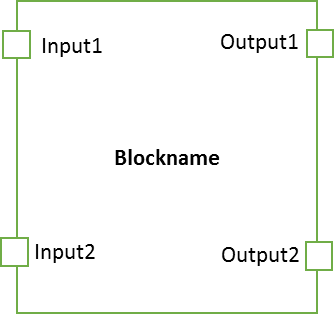
## Untyped Blocks (B)

An untyped block defines several **input** and **outputs** having a certain datatype. The implementation of the **algorithm** receives the input variables and calculates the output variables. The type is auto-generated by the loader of the file and is useful for implementations of types only used once.

The name of the type is the Blockname with an appendix of ‘Type’, but it cannot be reused by other types.

The block is defined in the following way:

B Blockname   
BI Input1 : Datatype  
BI Input2 : Datatype  
BO Output1 : Datatype  
BO Output2 : Datatype



This example creates an instance of a block with two inputs and two outputs. The inputs and outputs are also called ports. The block and the inputs and outputs are named. Each port name shall be unique within the given block, independent to the fact whether it is an input or an output.

Each port has a datatype that will be used for the implementation. The datatypes are listed in the specification below in chapter Port Data Types.

### Default values

The algorithm will receive the input values as a parameters. If the input value is wired to another output port, a warning will be given.

It is possible to set a default value, when there is no wiring. The syntax is a property of type ‘defaultvalue’. The defaultvalue will be casted to the datatype.

B Blockname   
BI Input1 : Datatype defaultvalue 5  
BI Input2 : Datatype defaultvalue 10  
BO Output1 : Datatype  
BO Output2 : Datatype

## Wire (W)

A wire connects the output of a block, which is an instance of a type, to an input of another block. It is not allowed to connect the output to an input within the block.

The value of the output will be transferred to the input of the next block. One output port may be connected to several input ports, but one input port may only be connected to one output port.

The syntax for a wire is the following:

W Blockname1.OutputPortName Blockname2.InputPortName

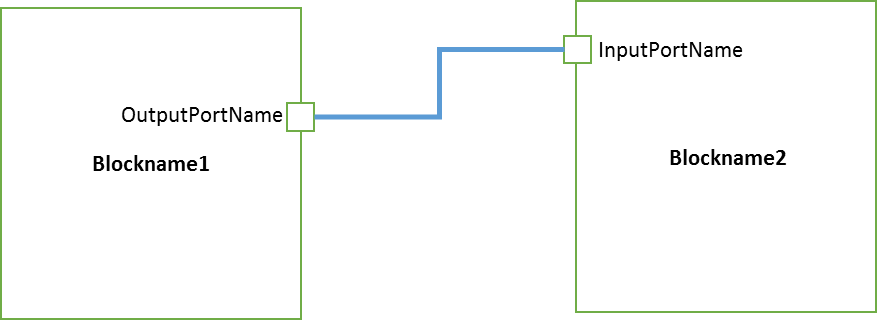


Image 8 - Example Wire

The wires also define a dependency of the blocks providing the data to another block. To avoid an undefined order of execution, no circular dependencies are allowed.

If you would like create a circular dependency, for closed loop controlling for example, you need to use a feedback wire.

## Feedback Wire (F)

To create a circular dependency of wires, you can use the feedback wire. The feedback wire will populate the values from input to output after all blocks have been executed.

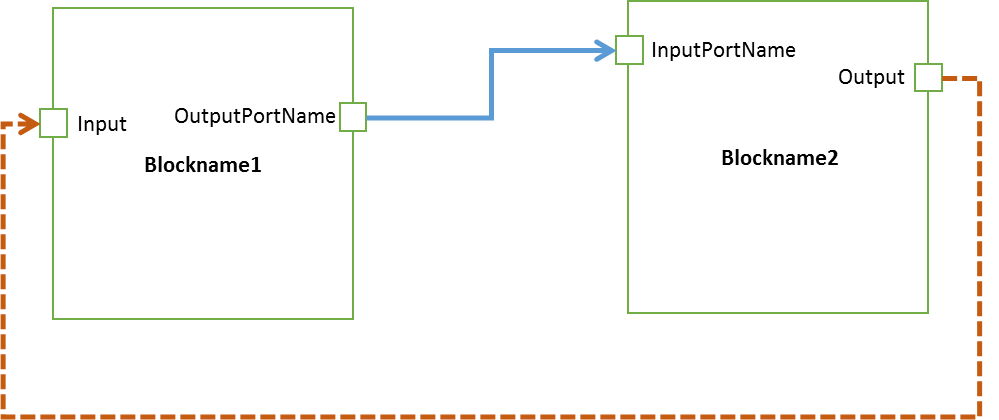


Image 9 - Example for feedback with wire

The feedback with wire can be configured in the same way as the normal wire, except the letter ‘F’ is used for the command.

F Blockname2.Output Blockname1.Input

## Type (T)

To improve the reusability of common blocks, a certain input/output scheme can be predefined as a type. The type will not be instantiated but can be used for typed blocks.

The syntax is the same as for the Blocks. The input and output ports are now prefixed with a T instead of a B.

T Typename   
TI Inputname1 : Datatype  
TI Inputname2 : Datatype  
TO Outputname1 : Datatype  
TO Outputname2 : Datatype

The names for the types and the blocks have to be unique within the whole project.

As an example for an adder:

T Adder  
TI Summand1 : Double  
TI Summand2 : Double  
TO Sum: Double

## Typed Block (B)

To ease the reuse of blocks and its algorithms, the blocks can be inherited by a certain type.

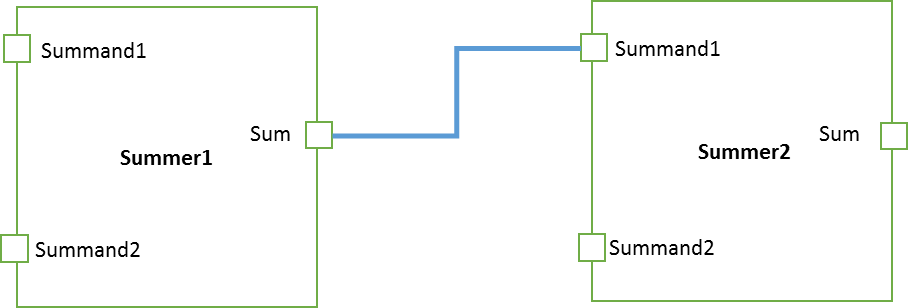
This can be done by the following definition:

B Blockname : Typename

The block will inherit all inputs, all outputs and its algorithm. A block can be instantiated multiple times. It is not possible to add additional inputs and outputs to the block.

As an example, the definition of two adders, which sum up three summands. The type definition as given above shall be assumed:

B Summer1 : Summer  
B Summer2 : Summer  
W Summer1.Sum Summer2.Summand1



It is possible to add a default value to an input of a typed block. This is done by adding a property to an existing input value:

B Summer1 : Summer

BI Summand1 defaultvalue 5

B Summer2 : Summer  
BI Summand1 defaultvalue 7

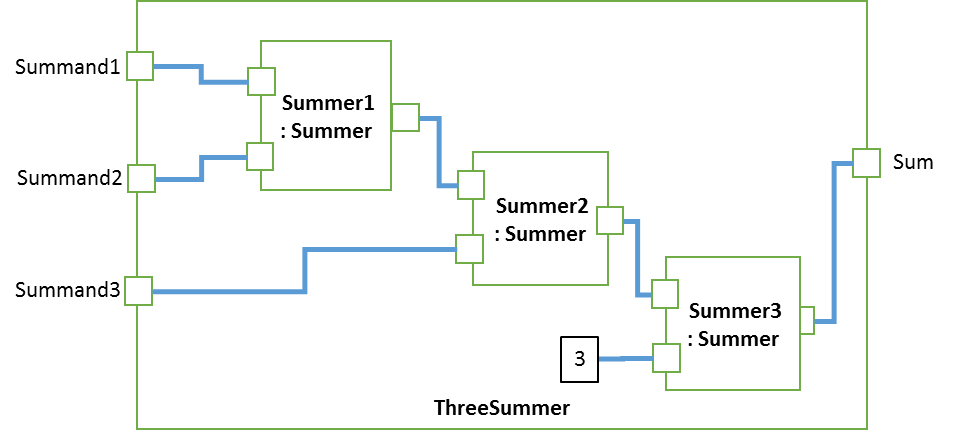
If the input is not existing in the type definition, an error will be thrown.

## Settings (S)

A setting line defines specific properties of a block.

This is not defined until now.

## Composite type (C)

Composite types define a new type where the algorithm is defined by interior blocks. One example is given in the following block which is able to sum up three input values and returns the sum of all three:

The definition for the example above would look like:

C ThreeSummer  
CB Summer1 : Summer  
CB Summer2 : Summer   
CB Summer3 : Summer

CI Summand1 : Double  
CI Summand2 : Double  
CI Summand3 : Double  
CO Sum : Double  
CW Summand1 Summer1.Summand1  
CW Summand2 Summer1.Summand2  
CW Summer1.Sum Summer2.Summand1  
CW Summand3 Summer2.Summand2  
CW Summer2.Sum Summer3.Summand1  
CBI Summer3.Summand2 defaultvalue 3 # Sets the default value

CW Summer3.Sum Sum

The CB lines defines the inner blocks. The inner blocks themselves can again be a composite block.

The CI lines define the input ports of the composite block. The CO lines define the output ports equivalently.

The CW lines define the inner wires. For ports of the composite block itself, no name needs to be given, for ports of the interior block, the ports name also needs to contain the blockname (*NameOfTheBlock*.*Portname*).

To use the block, it needs to be instantiated by a block.

B Summer1 : ThreeSummer

## Namespaces (N)

A new namespace can be defined in the following way:

N namespace

A namespace will be used for the creation of source files and puts all classes and structures in the given namespace.

The namespace is valid until the next namespace will be given. As an example:

N Logic

T And

T Or

*# Starts a new namespace*

N Algebra

T Sum

T Difference

The types „And“ and „Or“ will be put within the namespace “Logic”. The types “Sum” and “Difference” will be put in the the namespace “Algebra”.

To reference a certain namespace, the typename needs be prefixed with the name of the namespace and a dot.

B Combine : Logic.And

B Summer : Algebra.Sum

To use a type within a namespace, there is no need to prefix the typename.

B Combine : Logic.And

N Algebra

B Summer : Sum

The root element “\_”will be created within the current namespace at the end of the file.

## Technical commands

Every command, which start with an exclamation mark ‘!’ is a technical command, which is not needed for the model definition.

### Import of a file (!I)

To import another file, the file to be imported can be specified behind the !I statement:

!I otherfile.mbgi

The other file will be handled as if directly placed instead of the position of the import statement. This eases the use of multiple files. The path of the other file will be relative to the importing mbgi-file and not relative to the execution context of the application itself.

### Load a library (!L)

It is possible to load a library (.dll) file which was created by the model creator. The native and composite types of the library will be imported into the namespace of the source code creator and are made available.

!L ModellMeister.Library.dll

# Uses the type of the addition block and creates an instance

B Adder : Library.Algebra.Addition

When the C# file is compiled, it is necessary that the library is included as one of the referenced. In the current version of the loader, the types need to be referenced including the namespace.

A prefix “ModellMeister.” can be omitted, since it will be automatically created when necessary.

# Execution

Each type, whether anonymous type or explicit type, will be converted a structure. The structure will contain the input and output ports as explicit variables.

## The two phases

The simulation for each block will be done in two phases:

During the Fill phase, the input variables will be filled by the simulation framework. The input variables may be dependent on the wires from other outputs or default values.

The order of the filling is dependent to the dependency of the inputs to other outputs. Blocks which provide an output to another block, will be executed first. Only if all the input variables are populated by other blocks or default values, the block will be executed. This ensures a well-ordered execution of the context.

During the Execute phase, the algorithm will be called. The algorithm is responsible to set the output variables for each simulation step.

If an output variable needs to be propagated back, a feedback wire needs to be used. The feedback wire does not influence the order of execution.

## Properties for each simulation step

The simulation will receive the following information for each step:

|  |  |
| --- | --- |
| AbsoluteTime | The absolute time since the start of the simulation. |
| TimeInterval | The intervaltime since the last simulation step. |
| DoCancel | This value can be set, when the execution of the simulation shall be interrupted |

Depending on the settings of the execution environment, the model may be executed in real-time and TimeInterval describes the passed time between two calls. The other execution model would be via simulated input and outputs and the TimeInterval describes the simulated TimeInterval.

## Execution context

The created blocks are hosted into a global execution context.

The execution may also have input and output elements and these can also be wired from the inner blocks. Some examples of execution contexts will be found below.

### Simulation execution context

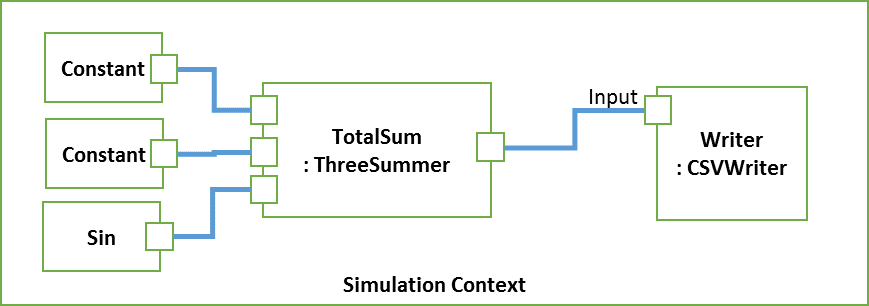
This context can be used for simulation.

The simulation will be started via:

mbsim.exe compiled.dll

The simulation will load the given .dll and look for the global composite element. This element will be instantiated and executed.

A simulation could look like the following one:



It takes two constants and a sine wave and returns the sum to the first output of the simulation. The generation file would look like the following (some type definitions for Constant, Sin and ThreeSummer are assumed):

C ThreeSummer  
… Definition of ThreeSummer as above

C \_ : SimulationContext  
  
T Constant  
TI InputValue  
TO Value  
  
T Sine  
TI Frequency  
TI Phase  
TO Value

T CSVWriter

TI Input  
  
B Const1 : Constant  
BI InputValue defaultvalue 5  
  
B Const2 : Constant  
BI InputValue defaultvalue 8  
  
B Sin : Sine  
BI Frequency defaultvalue 13.3  
BI Phase defaultvalue 1.57

B Writer : CSVWriter  
  
B TotalSum : ThreeSummer  
  
W Const1.Value TotalSum.Summand1  
W Const2.Value TotalSum.Summand2  
W Sin.Value TotalSum.Summand3  
W TotalSum.Sum Writer.Input

# Source code creation

The complete rules for the types, blocks and wirings can be stored in a single text file. The default extension for the text file is .mbgi. The file is called Model-Based-Generation Instruction file (MBGI-File).

The command line interface converts the MBGI- files into the appropriate source code files.

## C-Sharp Sourcecode

The following command line interface converts the MBGI-File to a C# Framework:

mbgi2cs.exe file.mbgi file.cs

This converts the file.mbgi to a set of classes in the C#-File file.cs.

For each type, a class will be created and the input and output ports will be modelled as properties of each class.

The algorithm of the type will be defined in the method *Execute*:

void Execute(ExecutionInfo info);

For non-composite blocks, the algorithm needs to be defined by the implementer. For composite blocks, the algorithm will be defined by the code generator itself.

The Init-Method is used to instantiate the inner blocks and is used to set the default values.

void Init();

# Full syntax

## Line

Each line describes an object.

{Type} {Name} ({PropertyType} {Content})\*

Whitespace symbols, like Space or Tab, are used to separate parts of a line definition. A line is separated by a \r, \n or \r\n.

A line always starts with a letter which describes the type of the object. A whitespace symbol, like Space or Tab, will be used to separate the letter to the following elements.

The second part is the name of the object as defined.

Properties to the object may follow. Each property definition consists a property type and the content.

Available PropertyTypes:

|  |  |
| --- | --- |
| : | Defines a type |
| defaultvalue | Defines a default value for an input |

Some examples:

|  |  |
| --- | --- |
| T adder | Type of name adder |
| B adderOne : adder | Block of name adderOne  The block is inherited of type adder. This is defined by the PropertyType ‘:’ and the content ‘adder’. |

## Comments

A comment starts with a “#”. The rest of the text within the line will be ignored during the following analysis.

## Object types

The following main object types exist:

|  |  |
| --- | --- |
| T | Defines a type |
| B | Defines a block |
| C | Defines a constant which can be used by other entities |
| W | Defines a wire between two blocks |
| # | Comment, this line will be skipped |

The following context dependent properties exist:

|  |  |
| --- | --- |
| BI | Input of a block |
| BO | Output of a block |
| TI | Input of a type |
| TO | Output of a type |
| CI | Input of a composite block |
| CO | Output of a composite block |
| W | Wire within a composite block or at global block. |

## Port Data Types

|  |  |
| --- | --- |
| Integer | Integer as defined in .Net-Datatype System.Int32 |
| Double | Double as defined in .Net-Datatype System.Double |
| String | String as defined in .Net-Datatype System.String |

# Implementation

The project is implemented in C# and is available in namespace *BurnSystems.ModelBase*. All dependencies are covered in the nuget Standard-Packages.

The following executables will be created:

|  |  |
| --- | --- |
| mbgi2cs.exe | Converts an .mbgi File to a C# file that can be used as a base |
| mbsim.exe | Simulates a library containing a model. |
|  |  |
|  |  |

## Full description of the command line tools

### mbgi2cs.exe

Reads in an MBGI file and writes a C# file which includes the information about the types, the blocks, the inputs, the outputs and the wires. The algorithm need to be implemented by the user.

Syntax: mbgi2cs.exe {mbgi-file} {cs-file}

The .cs file will be overwritten, if it already exists.

### mbsim.exe