Development of new units

Introduction

To start development of units the following conditions must be previously satisfied (refer to 'Configuration of VCProject.pdf'):

1. Installed Microsoft Visual Studio 2015 (Community).
2. Configured template project VCProject.

There are 4 different predefined templates of units available:

1. SteadyState: performs steady-state calculation; current state of such unit does not depend on the previous state, but only on the input parameters.
2. SteadyStateWithNLSolver: steady-state unit with connected internal solver of non-linear equations.
3. Dynamic: performs dynamic calculation; current state of this unit depends not only on the input parameters as well as on the previous state of the unit.
4. DynamicWithDAESolver: dynamic unit with connected internal solver of differential-algebraic equations.

1. Adding new unit to the template project

1. Copy the desired template of the unit from <*PathToSolution*>*\VCProject\UnitsTemplates* to the folder *Units* in solution (<*PathToSolution*>*\VCProject\Units*).
2. Rename template's folder according to the name of your new unit (further: ***<MyUnitFolder>***). The name can be chosen freely.
3. Rename project files in template's folder (**.vcxproj*, **.vcxproj.filters*) according to the name of the new unit.
4. Run the solution file (<*PathToSolution*>*\Dyssol.sln*) to open it in Visual Studio.
5. Add project with your new unit to the solution. To do this, select in Visual Studio [*File → Add → Existing Project*] and specify path to the project file (<*PathToSolution*>*\VCProject\Units\<MyUnitFolder>\<*.vcxproj>*).
6. Rename added project in Visual Studio according to the name of your unit.

Now one can implement functionality of new unit. To build your solution press F7, to run it in debug mode press F5. Files with new units will be placed to <*PathToSolution*>*\VCProject\Debug*.

As debug versions of compiled and built units contain a lot of additional information, which is used by Visual Studio to perform debugging, their calculation efficiency can be dramatically low. Thus, for the simulation purposes units should be built in Release mode:

2. Configuring Dyssol to work with implemented units

1. Build your units in release mode. To do this open your solution in Visual Studio (run file <*PathToSolution*>*\VCProject.sln*), switch *Solution configuration* combo box from the toolbox of Visual Studio from *Debug* to *Release* and build the project (press F7 or choose [*Build → Build project*] in program menu).
2. Configure Dyssol by adding the path to new units: run Dyssol, choose [Tools → *Models Manager*] and add path to your models (<*PathToSolution*>*\VCProject\Release*).

Now all new developed units will be available in *Dyssol*.

In general, usual configuration of *Models Manager* should include following path for units:
- <*InstallationPath*>*\Units* - list of standard units;
- <*PathToSolution*>*\VCProject\UnitsDebugLibs* – debug versions of standard units;
- <*PathToSolution*>*\VCProject\Debug* – debug versions of developed units;
- <*PathToSolution*>*\VCProject\Release* – release versions of developed units.

3.1. Development of steady-state units

Unit::Unit ()
Constructor of the unit: called only once when unit is added to the flowsheet. In this function a set of parameters should be specified:

1. Basic info:
- *m_sUnitName* – Name of the unit that will be displayed in Dyssol.
- *m_sAuthorName* – Unit's author
- *m_sUniqueID* – Unique identificator of the unit. Simulation environment distinguishes different units with the help of this identificator. You must ensure that ID of your unit is unique. This ID can be created manually or using GUID-generator of Visual Studio (*Tools->GUID Genarator*).
2. Specify ports: add new, rename or delete existing.
3. If unit has some additionally parameters, than specify them here.
4. Additional internal material streams can be defined here.
5. All other operations, which should take place only once during the unit's creation.

Unit::~Unit ()
Destructor of the unit: called only once when unit is removed from the flowsheet. Here all memory which has been previously allocated in the constructor should be freed.

Unit::Initialize(Time)
Unit's initialization. This function is called only once at the start of the simulation. Starting from this point, information about defined compounds, phases, distributions, etc. are available for the unit. Here you can create state variables and initialize some additionaly objects (for example additional material streams, state variables or plots).

Unit::Simulate(Time)
Steady-state calculation for a specified time point. This function is called iteratively for all time points for which this unit should be calculated. All main calculations should be implemented here.

Unit::Finalize()
Unit's finalization. This function is called only once at the end of the simulation. Here one can perform closing and cleaning operations to prepare for the next possible simulation run. Implementation of this function is not obligatory and can be skipped.

3.2. Development of steady-state units with internal NL solver

There is a possibility in the Dyssol system to solve nonlinear equation systems automatically. In this case, the unit should contain one or several additional objects of *NLModel* class. This class is used to describe NL systems and can be automatically solved with *NLSolver* class.

Unit::Unit()
Constructor of the unit: called only once when unit is added to the flowsheet. In this function a set of parameters should be specified:

1. Basic info:
- *m_sUnitName* – Name of the unit that will be displayed in Dyssol.
- *m_sAuthorName* – Unit's author
- *m_sUniqueID* – Unique identificator of the unit. Simulation environment distinguishes different units with the help of this identificator. You must ensure that ID of your unit is unique. This ID can be created manually or using GUID-generator of Visual Studio (*Tools->GUID Genarator*).
2. Specify ports: add new, rename or delete existing.
3. If unit has some additionally parameters, than specify them here.
4. Additional material streams can be defined here.
5. All other operations, which should take place only once during the unit's creation.

Unit::~Unit ()
Destructor of the unit: called only once when unit is removed from the flowsheet. Here all memory which has been previously allocated in the constructor should be freed.

Unit::Initialize(Time)
Unit's initialization. This function is called only once at the start of the simulation. Starting from this point, information about defined compounds, phases, distributions, etc. are available for the unit. Here you can create state variables and initialize some additionaly objects (for example holdups, material streams, state variables or plots).

In this function variables of all *NLModels* should be specified (using function *NLModel::AddNLVariable()*) and connection between *NLModel* and *NLSolver* classes should be created (by calling function *NLSolver::SetModel()*).

Unit::Simulate(Time)
Steady-state calculation for a specified time point. This function is called iteratively for all time points for which this unit should be calculated. All main calculations should be implemented here. Calculation of the defined NL-system can be run here by calling function *NLSolver::Calculate()*.

Unit::SaveState()
If the flowsheet contains recycled streams, than *SaveState*() function is called when the convergence on the current time interval is reached in order to have a possibility to return to the previous state of the unit if convergence failed during the calculation. Here all internal time-dependent variables which weren't added to the unit by using *AddStateVariable*() and *AddMaterialStream*() functions should be manually saved. Implementation of this function is not obligatory and can be skipped.

Unit::LoadState()
Load last state of the unit which has been saved with *SaveState*() function. Implementation of this function is not obligatory and can be skipped.

Unit::Finalize()
Unit‘s finalization. This function is called only once at the end of the simulation. Here one can perform closing and cleaning operations to prepare for the next possible simulation run. Implementation of this function is not obligatory and can be skipped.

NLModel::ResultsHandler()
Handling of results, which are returned from *NLSolver* on each time point. Called by solver every time when the solution in a new time point is ready.

NLModel::CalculateFunctions()
Here the NL system should be specified. This function will be called by solver automatically.

3.3. Development of dynamic units

Unit::Unit ()
Constructor of the unit: called only once when unit is added to the flowsheet. In this function a set of parameters should be specified:

1. Basic info:
- *m_sUnitName* – Name of the unit that will be displayed in Dyssol.
- *m_sAuthorName* – Unit's author
- *m_sUniqueID* – Unique identificator of the unit. Simulation environment distinguishes different units with the help of this identificator. You must ensure that ID of your unit is unique. This ID can be created manually or using GUID-generator of Visual Studio (*Tools->GUID Genarator*).
2. Specify ports: add new, rename or delete existing.
3. If unit has some additionally parameters, than specify them here.
4. Internal holdups and additional material streams can be defined here.
5. All other operations, which should take place only once during the unit's creation.

Unit::~Unit ()
Destructor of the unit: called only once when unit is removed from the flowsheet. Here all memory which has been previously allocated in the constructor should be freed.

Unit::Initialize(Time)
Unit's initialization. This function is called only once at the start of the simulation. Starting from this point, information about defined compounds, phases, distributions, etc. are available for the unit. Here you can create state variables and initialize some additionaly objects (for example holdups, material streams or state variables).

Unit::Simulate(Tstart, Tend)
Dynamic calculation of the unit on a specified time interval. All logic of the unit's model must be implemented here.

Unit::SaveState()
If the flowsheet contains recycled streams, than *SaveState*() function is called when the convergence on the current time interval is reached in order to have a possibility to return to the previous state of the unit if convergence failed during the calculation. Here all internal time-dependent variables which weren't added to the unit by using *AddStateVariable*(), *AddMaterialStream*() or *AddHoldup*() functions should be manually saved. Implementation of this function is not obligatory and can be skipped.

Unit::LoadState()
Load last state of the unit which has been saved with the *SaveState*() function. Implementation of this function is not obligatory and can be skipped.

Unit::Finalize()
Unit's finalization. This function is called only once at the end of the simulation. Here one can perform closing and cleaning operations to prepare for the next possible simulation run. Implementation of this function is not obligatory and can be skipped.

3.4. Development of dynamic units with internal DAE solver

There is a possibility in the Dyssol system to solve systems of differential-algebraic equations automatically. In this case, the unit should contain one or several additional objects of *DAEModel* class. This class is used to describe DAE systems and can be automatically solved with *DAESolver* class.

Unit::Unit()
Constructor of the unit: called only once when unit is added to the flowsheet. In this function a set of parameters should be specified:

6. Basic info:
- *m_sUnitName* – Name of the unit that will be displayed in Dyssol.
- *m_sAuthorName* – Unit's author
- *m_sUniqueID* – Unique identificator of the unit. Simulation environment distinguishes different units with the help of this identificator. You must ensure that ID of your unit is unique. This ID can be created manually or using GUID-generator of Visual Studio (*Tools->GUID Genarator*).

7. Specify ports: add new, rename or delete existing.
8. If unit has some additionally parameters, than specify them here.
9. Internal holdups and additional material streams can be defined here.
10. All other operations, which should take place only once during the unit's creation.

Unit::~Unit ()
Destructor of the unit: called only once when unit is removed from the flowsheet. Here all memory which has been previously allocated in the constructor should be freed.

Unit::Initialize(Time)
Unit's initialization. This function is called only once at the start of the simulation. Starting from this point, information about defined compounds, phases, distributions, etc. are available for the unit. Here you can create state variables and initialize some additionaly objects (for example holdups, material streams or state variables).

In this function variables of all *DAEModels* should be specified (using function *DAEModel::AddDAEVariable()*) and connection between *DAEModel* and *DAESolver* classes should be created (by calling function *DAESolver::SetModel()*).

Unit::Simulate(Tstart, Tend)
Dynamic calculation for a specified time interval. Is called for each time window on simulation interval. Calculation of the defined DAE-system can be run here by calling function *DAESolver::Calculate()*.

Unit::SaveState()
If the flowsheet contains recycled streams, than *SaveState*() function is called when the convergence on the current time interval is reached in order to have a possibility to return to the previous state of the unit if convergence failed during the calculation. Here all internal time-dependent variables which weren't added to the unit by using *AddStateVariable*(), *AddMaterialStream*() or *AddHoldup*() functions should be manually saved. Implementation of this function is not obligatory and can be skipped.

Unit::LoadState()
Load last state of the unit which has been saved with *SaveState*() function. Implementation of this function is not obligatory and can be skipped.

Unit::Finalize()
Unit‘s finalization. This function is called only once at the end of the simulation. Here one can perform closing and cleaning operations to prepare for the next possible simulation run. Implementation of this function is not obligatory and can be skipped.

DAEModel::ResultsHandler()
Handling of results, which are returned from *DAESolver* on each time point. Called by solver every time when the solution in a new time point is ready.

DAEModel::CalculateResiduals()
Here the DAE system should be specified in implicit form. This function will be called by solver automatically.

4. Configuring unit to work with MATLAB

There is a possibility to use MATLAB Engine API in the Dyssol during the development of units. It requires an installed **32-bit version** of MATLAB. For API description refer to following links:

- http://de.mathworks.com/help/matlab/calling-matlab-engine-from-c-c-and-fortran-programs.html
- http://de.mathworks.com/help/matlab/cc-mx-matrix-library.html

To enable interaction with MATLAB configure template project with your unit:

1. Add a new environment variable in Windows with the path to the MATLAB installation directory: *Computer* → *Properties* → *Advanced system settings* → *Environment variables* → *System variables* → *New* → Variable Name: *MATLAB_PATH*; Variable value: path to installed 32-bit version of MATLAB (e.g. *C:\Program Files (x86)\MATLAB\R2014b*). It may require restarting the Visual Studio or computer to apply changes.
2. Provide the main project of template solution with path to MATLAB libraries: select project *ModelsAPI* in solution explorer, then choose [*Project* → *Properties* → *Configuration Properties* → *Environment*], set combo box *Configuration* in the top of the window to position *All Configurations* and provide the *Environment* field with parameter *PATH=$(MATLAB_PATH)\bin\win32*.
3. Provide unit's project with the path to MATLAB libraries: select project with your unit in solution explorer, then choose [*Project* → *Properties* → *Configuration Properties* → *Environment*], set combo box *Configuration* in the top of the window to position *All Configurations* and provide the *Environment* field with parameter *PATH=$(MATLAB_PATH)\bin\win32*.
4. Add MATLAB libraries to the unit's project: select project with your unit in solution explorer, then choose [*Project* → *Properties* → *Configuration Properties* → *Linker* → *Input* → *Additional Dependencies*], set combo box *Configuration* in the top of the window to position *All Configurations* and add following four libraries at the beginning of the input field: *libmx.lib;libmat.lib;libeng.lib;libmex.lib;*
5. Insert MATLAB's header in Unit.h: add the line `#include "engine.h"` to the include section at the top of your Unit.h file.