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.

Interfaces of DAE model:

Constructors
CDAEModel
Functions to work with variables
AddDAEVariable GetVariablesNumber ClearVariables
Functions to work with tolerances
SetTolerance GetRTol GetATol
Virtual functions which should be overriden in inherited classes
CalculateResiduals ResultsHandler
Other functions
Clear SetUserData

Constructors

CDAEModel *(void)*
Basic constructor. Creates an empty DAE model.

Functions to work with variables

unsigned AddDAEVariable *(bool _bIsDifferentiable, double _dVariableInit, double _dDerivativeInit, double _dConstraint = 0.0)*
Adds new algebraic (*_bIsDifferentiable* = *false*) or differential (*_bIsDifferentiable* = *true*) variable with initial values *_dVariableInit* and *_dDerivativeInit*. Should be called in function *Initialize()* of the unit. Returns unique index of added variable. *_dConstraint* sets the constraint for the variable:

- 0.0: no constraint
- 1.0: Variable ≥ 0.0
- −1.0: Variable ≤ 0.0
- 2.0: Variable > 0.0
- −2.0: Variable < 0.0

unsigned GetVariablesNumber *()*
Returns current number of defined variables.

void ClearVariables *()*
Removes all defined variables from the model.

Functions to work with tolerances

void SetTolerance *(double _dRTol, double _dATol)*
Sets values of relative and absolute tolerances for all variables.

void SetTolerance *(double _dRTol, std::vector< double > &_vATol)*
Sets value of relative tolerance for all variables and absolute tolerances for each variable separately.

double GetRTol *()*
Returns current relative tolerance.

double GetATol *(unsigned _dIndex)*
Returns current absolute tolerance for specified variable.

Virtual functions which should be overriden in inherited classes

virtual void CalculateResiduals *(double _dTime, double *_pVars, double *_pDerivs, double *_pRes, void *_pUserData)*
Computes the problem residual for given values of the independent variable *_dTime*, state vector *_pVars*, and derivatives *_pDerivs*. Here the DAE system should be specified in implicit form. Function will be called by solver automatically.

_dTime	– Current value of the independent variable t.
_pVars	– Pointer to an array of the dependent variables, $y(t)$.
_pDerivs	– Pointer to an array of derivatives $y'(t)$.
_pRes	– Output residual vector $F(t, y, y')$.
_pUserData	– Pointer to user's data. Is used to provide access from this function to unit's data.

virtual void ResultsHandler *(double _dTime, double *_pVars, double *_pDerivs, void *_pUserData)*
Processing the results returned by the solver at each calculated step. Called by solver every time when the solution in new time point is ready.

_dTime	– Current value of the independent variable t.
_pVars	– Current values of the dependent variables, $y(t)$.
_pDerivs	– Current values of derivatives $y'(t)$.
_pUserData	– Pointer to user's data. Is used to provide access from this function to unit's data.

Other functions

void Clear *()*
Removes all data from the model.

void SetUserData *(void *_pUserData)*
Set pointer to user's data. This data will be returned in overloaded functions *CalculateResiduals()* and *ResultsHandler()*. Usually is used to provide access from these functions to unit's data.

Interfaces of DAE solver:

Constructors
CDAESolver
Functions to work with model
SetModel SetMaxStep Calculate
Other functions
SaveState LoadState GetError

Constructors

CDAESolver *(void)*
Basic constructor. Creates an empty solver.

Functions to work with model

bool SetModel *(CDAEModel *_pModel)*
Sets model to a solver. Should be called in function *Initialize()* of the unit. Returns *false* on error.

bool SetMaxStep *()*
Sets maximum time step for solver. Should be used in *Unit::Initialize()* before the function *CDAESolver::SetModel()*.

bool Calculate *(double _dTime)*
Solves problem on a given time point. Should be called in function *Simulate()* of the unit. Returns *false* on error.

bool Calculate *(double _dStartTime, double _dEndTime)*
Solves problem on a given time interval. Should be called in function *Simulate()* of the unit. Returns *false* on error.

Other functions

void SaveState *()*
Saves current state of the solver. Should be called in function *SaveState()* of the unit.

void LoadState *()*
Loads last saved state of the solver. Should be called in function *LoadState()* of the unit.

std::string GetError *()*
Returns error's description. Can be called if function *SetModel()* or *Calculate()* returns *false*.

Example:

Assume that it is necessary to solve the system of differential-algebraic equations:

$$\begin{cases} \dfrac{dx}{dt} = -0.04 \cdot x + 10^4 \cdot y \cdot z & x(0) = 0.04 \\ \dfrac{dy}{dt} = 0.04 \cdot x - 10^4 \cdot y \cdot z - 3 \cdot 10^7 \cdot y^2 & y(0) = -0.04 \\ x + y + z = 1 & \end{cases}$$

where x, y, z are fractions of solid liquid and vapor phases of the output stream of the unit. Development of the unit for automatic calculation of this DAE by using of built-in solver of Dyssol can be done in few steps:

1. Add new template unit *DynamicUnitWithSolver* to your solution (refer to the file 'Units development.pdf').
2. In file Unit.h is already defined a class of DAE model *CMyDAEModel* with two functions, which must be overridden (*CalculateResiduals()* and *ResultsHandler()*) and class of unit *CUnit* with two additional variables: for DAE model (*CMyDAEModel m_Model*) and DAE solver (*CDAESolver m_Solver*).

   Add three variables in class *CMyDAEModel* to store indexes of the state variables of equation system (for x, y and z variables). Call them *m_nS*, *m_nL* and *m_nV* respectively. After adding description of class *CMyDAEModel* should look like this:

```
class CMyDAEModel : public CDAEModel
{
public:
	unsigned m_nS; // solid fraction
	unsigned m_nL; // liquid fraction
	unsigned m_nV; // vapor fraction
public:
	void CalculateResiduals( double _dTime, double* _pVars,
	                    double* _pDers, double* _pRes, void* _pUserData );
	void ResultsHandler( double _dTime, double* _pVars,
	                    double* _pDerivs, void *_pUserData );
};
```

3. Setup unit's basic info (name, author's name, unique id, ports) as it described in 'Units development.pdf'. At the end provide model with pointer to your unit to have an access to the unit from functions of the model. Unit's constructor after that must look as follows:

```
CUnit::CUnit()
{
	m_sUnitName = "DummyUnit4";
	m_sAuthorName = "Author";
	m_sUniqueID = "344BCC0048AA4c3a9117F20A9F8AF9A8";

	AddPort( "InPort", INPUT_PORT );
	AddPort( "OutPort", OUTPUT_PORT );

	m_Model.SetUserData( this );
}
```

4. Implement *Initialize()* function of the unit: add 3 variables with specified initial conditions to the model (using *AddDAEVariable()*) according to the equation system. As initials use phase fractions of the input stream.

```
m_Model.m_nS=m_Model.AddDAEVariable(true,dSolidFraction,0.04,1.0);
m_Model.m_nL=m_Model.AddDAEVariable(true,dLiquidFraction,-0.04,1.0);
m_Model.m_nV=m_Model.AddDAEVariable(false,dVaporFraction,0,1.0);
```

Since *Initialize()* function is called every time, when simulation starts, all variables must be previously removed from the model by calling *ClearVariables()*.

```
m_Model.ClearVariables();
```

Set tolerances to the model (*SetTolerance()* function). As tolerances for the model global tolerances of the system can be used.

```
m_Model.SetTolerance( GetRelTolerance(), GetAbsTolerance() );
```

Now model can be connected to the solver by calling *SetModel()*. To have a possibility to receive errors from solver, connect it to the global errors handling procedure.

```
if( !m_Solver.SetModel( &m_Model ) )
	RaiseError( m_Solver.GetError() );
```

Function *Initialize()* of the unit after all changes:

```
void CUnit::Initialize( double _dTime )
{
	CMaterialStream *pInpStream = GetPortStream( "InPort" );

	double dSFr = pInpStream->GetSinglePhaseProp( _dTime, FRACTION, SOA_SOLID );
	double dLFr = pInpStream->GetSinglePhaseProp( _dTime, FRACTION, SOA_LIQUID );
	double dVFr = pInpStream->GetSinglePhaseProp( _dTime, FRACTION, SOA_VAPOR );

	m_Model.ClearVariables();
	m_Model.m_nS = m_Model.AddDAEVariable( true, dSFr, 0.04, 1.0 );
	m_Model.m_nL = m_Model.AddDAEVariable( true, dLFr, -0.04, 1.0 );
	m_Model.m_nV = m_Model.AddDAEVariable( false, dVFr, 0, 1.0 );

	m_Model.SetTolerance( GetRelTolerance(), GetAbsTolerance() );
	if( !m_Solver.SetModel( &m_Model ) )
		RaiseError( m_Solver.GetError() );
}
```

5. Connect solver to a system saving/loading procedure in functions *SaveState()* and *LoadState()* of the unit:

```
void CUnit::SaveState()
{
	m_Solver.SaveState();
}

void CUnit::LoadState()
{
	m_Solver.LoadState();
}
```

6. In function *Simulate()* of the unit calculation procedure should be run by calling function *Calculate()* of the solver. Additionally solver can be connected to the system's errors handling procedure to receive possible errors during the calculation. Unit's *Simulate()* function after that must look as follows:

```
void CUnit::Simulate( double _dStartTime, double _dEndTime )
{
	CMaterialStream *pInputStream = GetPortStream( "InPort" );
	CMaterialStream *pOutputStream = GetPortStream( "OutPort" );

	pOutputStream->RemoveTimePointsAfter( _dStartTime, true );
	pOutputStream->CopyFromStream( pInputStream, _dStartTime );

	if( !m_Solver.Calculate( _dStartTime, _dEndTime ) )
		RaiseError( m_Solver.GetError() );
}
```

7. In *CalculateResiduals()* function of the model DAE system in implicit form should be described:

$$\begin{cases} x' - (-0.04 \cdot x + 10^4 \cdot y \cdot z) = 0 \\ y' - (0.04 \cdot x - 10^4 \cdot y \cdot z - 3 \cdot 10^7 \cdot y^2) = 0 \\ x + y + z - 1 = 0 \end{cases}$$

```
void CMyDAEModel::CalculateResiduals( double _dTime, double* _pVars,
                                      double* _pDers, double* _pRes, void* _pUserData )
{
    _pRes[m_nS]=_pDers[m_nS]-(-0.04*_pVars[m_nS]+1.0e4*_pVars[m_nL]*_pVars[m_nV]);
    _pRes[m_nL] = _pDers[m_nL] - ( 0.04*_pVars[m_nS] -
             1.0e4*_pVars[m_nL]*_pVars[m_nV] - 3.0e7*_pVars[m_nL]*_pVars[m_nL] );
    _pRes[m_nV] = _pVars[m_nS] + _pVars[m_nL] + _pVars[m_nV] - 1;
}
```

8. Last step is handling of results from the solver (*CMyDAEModel::ResultsHandler*). Calculated fractions can be set here to the output stream of the unit. To access to the unit's data previously set pointer *_pUserData* can be used:

```
void CMyDAEModel::ResultsHandler( double _dTime, double* _pVars, double*
                                                    _pDerivs, void *_pUserData )
{
	CUnit *unit = static_cast<CUnit*>(_pUserData);
	CMaterialStream *pStream =
		static_cast<CMaterialStream*>(unit->GetPortStream("OutPort"));

	pStream->AddTimePoint( _dTime );

	pStream->SetSinglePhaseProp( _dTime, FRACTION, SOA_SOLID, _pVars[m_nS] );
	pStream->SetSinglePhaseProp( _dTime, FRACTION, SOA_LIQUID, _pVars[m_nL] );
	pStream->SetSinglePhaseProp( _dTime, FRACTION, SOA_VAPOR, _pVars[m_nV] );
}
```