

# LWC12 – MetaEdit+

Version 1.0, Feb 25, 2012

Based on LWC 2012 Assignment, November 2, 2011

Juha-Pekka Tolvanen  
jpt@metacase.com

---

## Abstract

MetaEdit+ is a mature language workbench that supports graphical diagram, matrix and table representations. We demonstrate it by completing the tasks of the 2012 Language Workbench Challenge from <http://languageworkbenches.net>.

---

## Getting started

You can obtain a beta version of MetaEdit+ 5.0 from [www.metacase.com/download](http://www.metacase.com/download). Follow the instructions in the email and below to install MetaEdit+. Do not log in yet.

After installation you will have in your selected MetaEdit+ directory:

- testmep50Build\*.exe: MetaEdit+ program
- demo\*: the demo database
- cairo.dll: Cairo graphics libraries
- What's New \*.txt: the list of new features, improvements and fixes
- MetaEdit+ 5.0 Beta Primer.pdf

Clicking the .exe starts MetaEdit+ 5.0 beta, but **do not log in to the demo repository**: we want to get the LWC12 repository.

The repository for the LWC tasks can be downloaded from:

- <http://www.metacase.com/support/50/repository/LWC12.zip>

Unzip the LWC12 repository directory structure into your MetaEdit+ working directory, e.g. My Documents\MetaEdit+ 5.0.

In the MetaEdit+ Startup Launcher, press F5 to update the repository list, which should now show LWC12 as well as the standard demo repository. (If you can't see it, make sure you unzipped correctly, giving paths like My Documents\MetaEdit+ 5.0\LWC12\manager.ab.)

Select the LWC12 repository, and you will see the projects it contains in the list on the right.

- P&I: Piping and Instrumentation languages implemented for LWC2012
- Home: A design of home using the LWC2012 languages

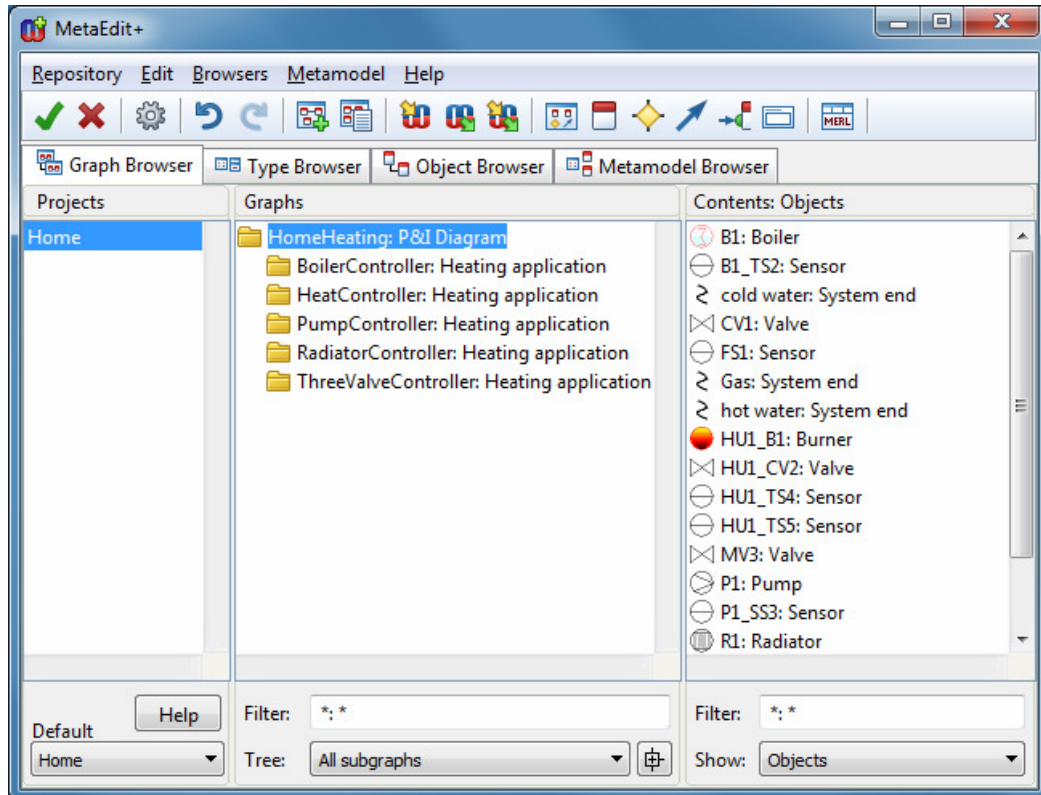
Select 'Home' project and press Login to open the models of LWC2012 home heating system.

The rest of this document describes the language and generators for the LWC12 challenge as well as how they were developed with MetaEdit+. If you want to follow the DSL implementation use the metamodeling and modeling tools of MetaEdit+.

## Piping and Instrumentation support

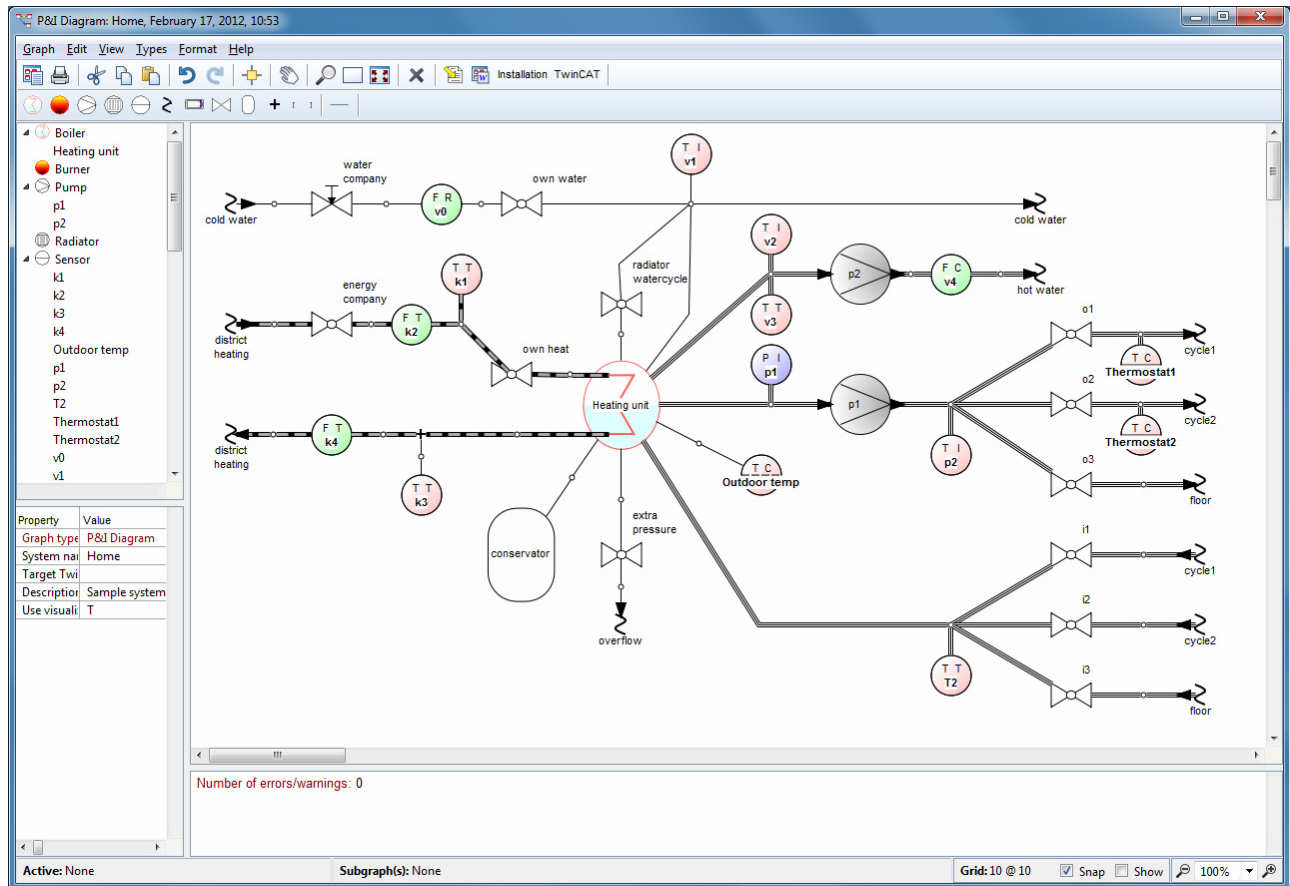
For modeling heating applications the MetaEdit+ supported modeling solution provides two integrated languages and a number of generators. In addition to editors, MetaEdit+ provides also various browsers, predefined generators, multi-user support, etc [1].

The main window for browsing the models and accessing the editors and generators is shown below. The opened 'Home' project contains one P&I Diagram and five heating application controllers.



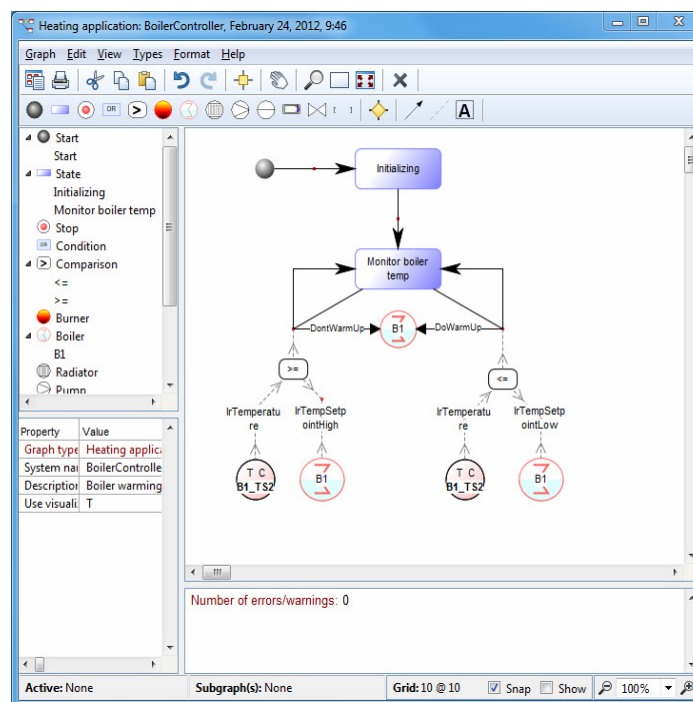
### 1.1 Piping language for structure

P&I Diagram specifies pipe connections among the various instruments. Those instruments which have controllable behavior are specified in a subdiagram using another DSL.



## 1.2 Control behavior

Heating application language specifies the control logic of the system: states of the controller, conditions based on instrument data and various actions to control the instruments. The instruments are the same as in the P&I Diagram and they can be accessed from both type of diagrams.



### 1.3 Generators

Generators are provided for various purposes:

- 'expFiles' produces the code in the format of TwinCAT exp.files. The generated code provides the control logic blocks, its simulation as well as additional resources, like datatypes and task structures. The generated code follows the same style and conventions as the reference implementation.
- 'TwinCAT' provides integration with TwinCAT PLC control tool: the code, as produced by 'expFiles' is imported into TwinCAT PLC control tool, migrated with the platform providing the basic building blocks and compiled for execution/simulation.
- 'Installation' produces HW installation guide in HTML listing the type of instruments and the amount of pipe needed (calculated from the length of individual pipes).
- 'Doc' generates documentation of the system into a Word document.

In addition to these generators other generators are defined for model checking (shown in the bottom of Diagram Editor), generating interface descriptions as well as producing textual description of the piping for those who prefer text instead of diagrams.

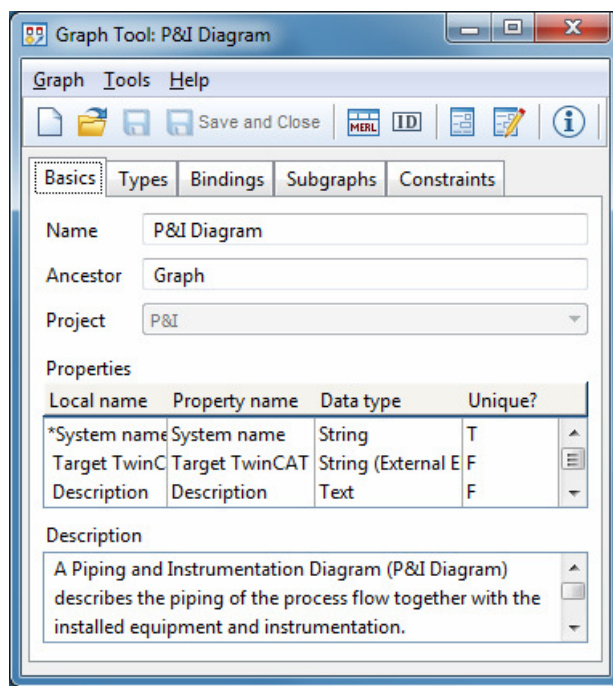
After the metamodel, as above, is drawn in Diagram Editor of MetaEdit+ it can be instantiated and tested immediately with the modeling editors providing full editor functionality (copy/paste (special...), undo, replace, trace, print etc.). Once the DSL is complete it can be given for the developers using MetaEdit+ Modeler.

## Form-based metamodeling tools

Form-based metamodeling tools define all aspects of the language: not only abstract syntax, but also language rules, notation and generators for model checking, code generation, documentation generation etc. The form-based tools are integrated allowing to access and trace between all the language elements: e.g. from abstract syntax to notational symbols, from generator definition to metamodel, from debugged generator to models etc.

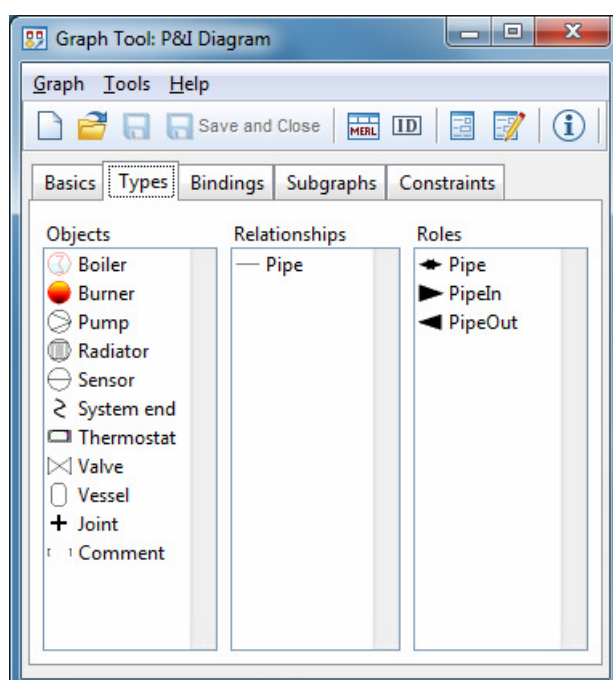
Most importantly, the definition of DSL elements is automatically applied in the various modeling editors (Diagram, Matrix, Table), browsers (Graph, Object, Type) and generators. This supports agile and incremental DSL definition: updates to the DSL can be tested immediately and shown for the language users.

### Language concepts (abstract syntax)



Graph tool defines the individual languages. Here a definition for a 'P&I Diagram' is given. A diagram itself has properties, and each property has a more detailed definition. For example, 'System name' is an identifying (marked with '\*') property. It is of string data type and its value must be unique: there can't be other P&I Diagrams with the same name.

The Graph Tools includes also a description field to document the language. The description given is used in the language help available in the modeling editors (Help | Graph Type...).



Types tab in the Graph Tool shows the individual language concepts: object types, relationship types and role types. These different type of concepts can be here added or removed from the language.



**Object Tool: Sensor**

Object Tools Help

Name:

Ancestor:

Project:

Local name	Property name	Data type	Unique?
*Name	Sensor name	String	F
Mounting	Mounting	String (Fixed List)	F
Measured varial	Measured variable	String (Fixed List)	F
Function	Function	String (Overridable)	F

Description: Measure physical characteristics of the gas or fluid stream (e.g. pressure, temperature, flow). Sensor unit can connected to the pipe with one or more connection points. Flow sensors need two connection points, others need one.

For each type, a form-based metamodeling tool shows its definition. Here Object Tool shows the definition of a 'Sensor': its name, ancestor type, four different property types and documenting description.

Other kind of types, like property types, port types, role types and relationship types are defined similarly with the form-based metamodeling tools.

**Property Tool: Sensor name for Sen...**

Property Tools Help

Name:

Ancestor:

Project:

Datatype:  Edit

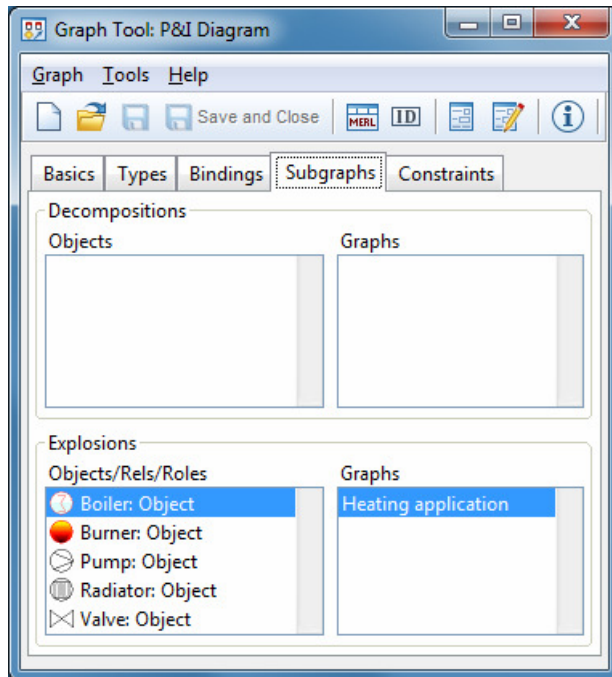
Widget:

Default Value:

Value Regex:

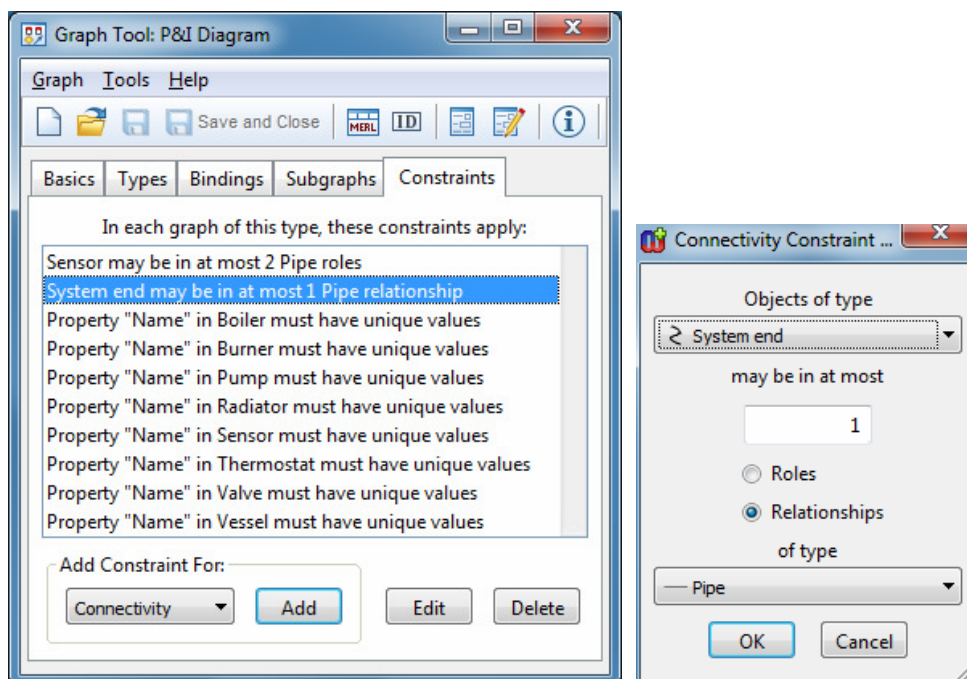
Description: Name of the sensor.

A definition of 'Sensor name' is shown in Property Tool. It is of String data type, uses Input Field for entry, does not have any default value and must have a mandatory value. The constraint on mandatory value is specified using the regular expression ('.+').



Subgraph tab in Graph Tool sets the links between the graphs created. The linked graphs may be of different type as in case of LWC2012: A behavior of a Pump for example can be described in another kind of a graph (called here 'Heating application').

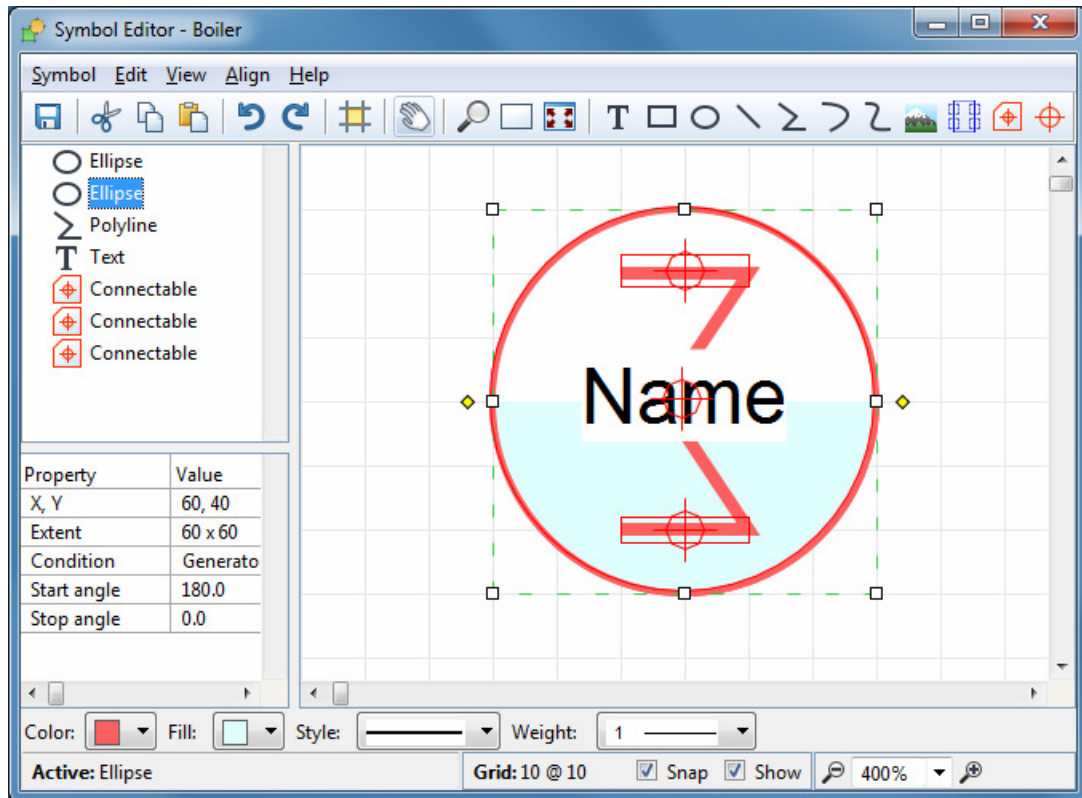
Constraints are specified with the form-based tools as a part of the metamodel or defined with the generators. Below some implemented constraints are shown for the P&I Diagram. The constraints are given as data and entered by choosing from the existing set of constraint types. For example, 'System end' may be connected to one 'Pipe' relationship only.



## Notation

The notation of the DSL is defined with Symbol Editor. It can be opened from the form-based tools while defining the language abstract syntax. Below Symbol Editor shows the symbol for 'Boiler' Object Type.





The notational symbol of 'Boiler' consists of multiple symbol elements, and each symbol element can be drawn here, selected from the symbol library, or retrieved from other elements used in the models. Individual symbol elements can be made conditional depending on the model data.

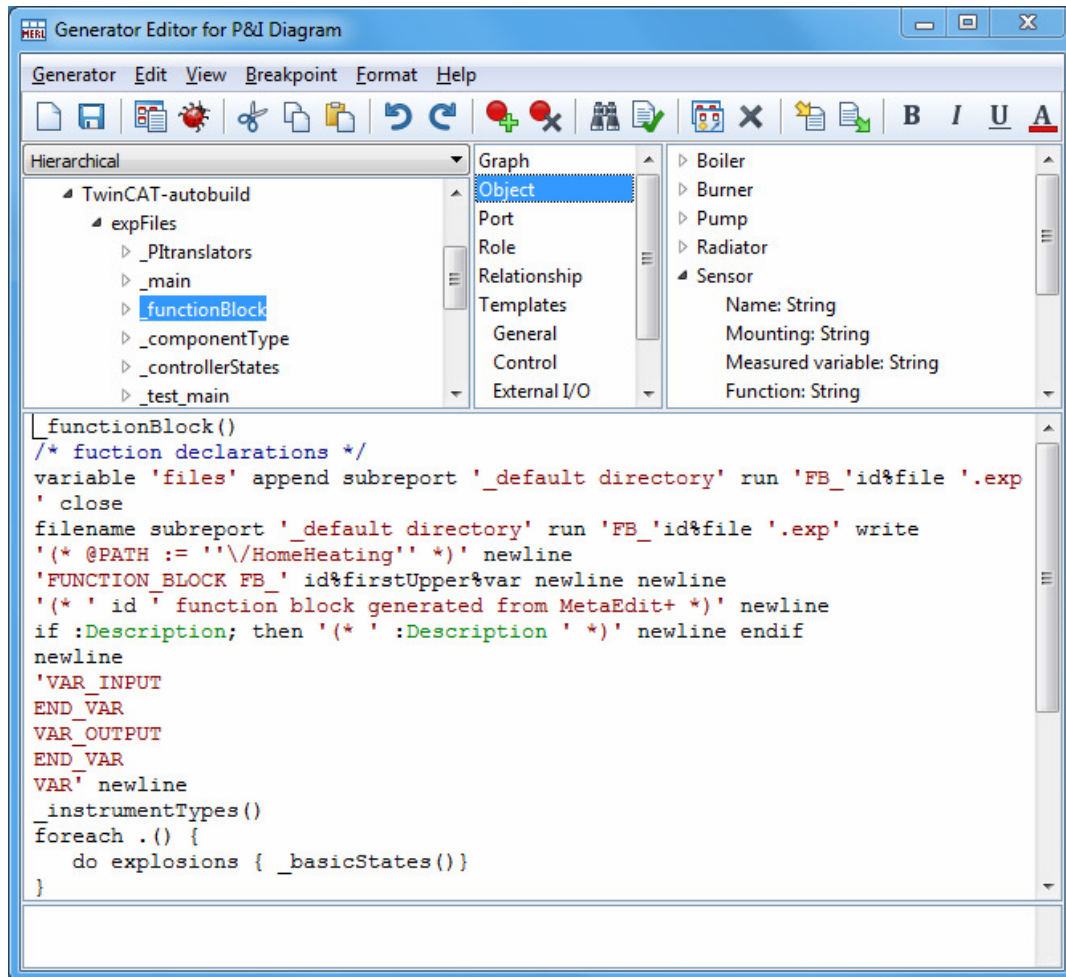
The symbol of 'Boiler' consists of two ellipses, a polyline, and a text property showing the name of the boiler. The symbol definition also shows three different kinds of connectables: how bindings can be drawn connecting the 'Boiler' with a 'Pipe' to other instruments. 'Boiler' has two heat connectables specifying 'Pipe' connections for hot (input) and cold (output). The third main connectable is used to draw connections for instruments using the heated water, liquid etc.

Symbol Editor allows importing existing symbols, or their parts, as vector graphics or bitmaps. Symbol Editor may also retrieve the symbols from the in-built Symbol Library of MetaEdit+.

### Generators

MetaEdit+ Generator System is used to define generators for various needs. In case of LWC2012 assignment, generators are used to produce code, model checking, installation guide in HTML, Word documentation of the system, as well as importing the generated code directly into TwinCat environment.

MERL, a domain-specific language for generator development, is used to define generators. Figure below shows the MERL definition for function block generation in Generator Editor for P&I Diagram. The generator script is shown bottom of the window, a hierarchy of generator modules on the top-left, MERL templates in the top-middle, and DSL elements (metamodel) on the top-right compartment of the window.



Generator Editor supports syntax highlighting, static code analysis, error detection, navigation among the subgenerators and content assistant showing the metamodel structures while writing the generator. MetaEdit+ provides also full MERL source-level debugger with breakpoints, interrupts, conditional breakpoints, live editing of variables etc. In addition to the Generator System, MetaEdit+ offers API based on webservices/.Net/SOAP for accessing the model elements using other generator systems.

The form-based metamodeling tools also support definition of user-interface elements (toolbar and browser icons, property editing dialogs and visible element identifiers) in case the ones provided by default are not considered adequate by the language engineer.

## 2 Define a P&I network of the central heating system

At any point of time during the DSL definition, created languages can be tried out in MetaEdit+ using the ready editors, browsers, multi-user support, printing functions etc. This enables fast prototyping and incremental development of the DSL.

Diagram below follows closely the visualization proposed in the LWC2012 assignment. For example, pipes which are marked to be thermally insulated are shown with thick lines, pipes which are jacketed are shown with double lines, and pipes which don't have any cover are shown with thin solid lines. This visualization was defined with the Symbol Editor.

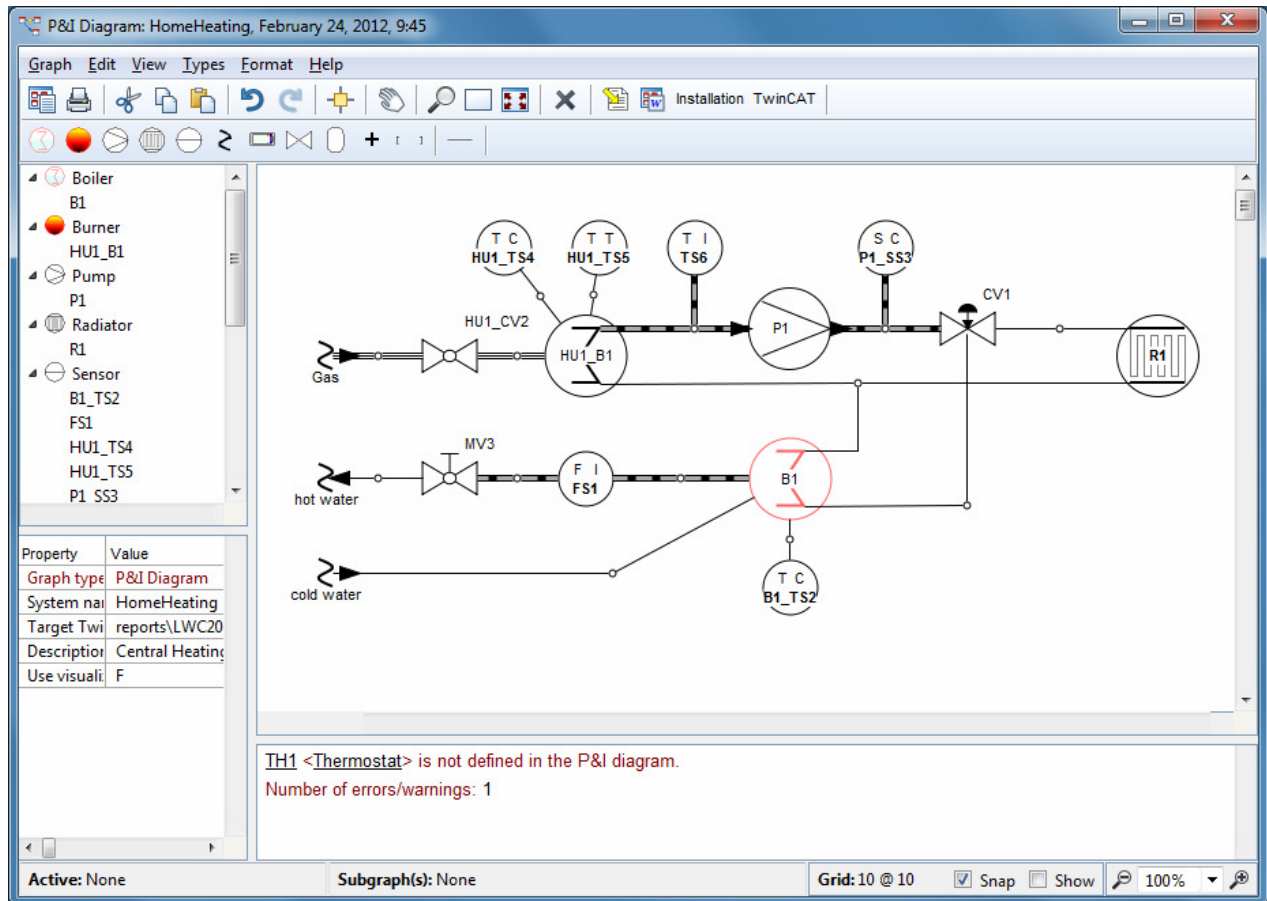
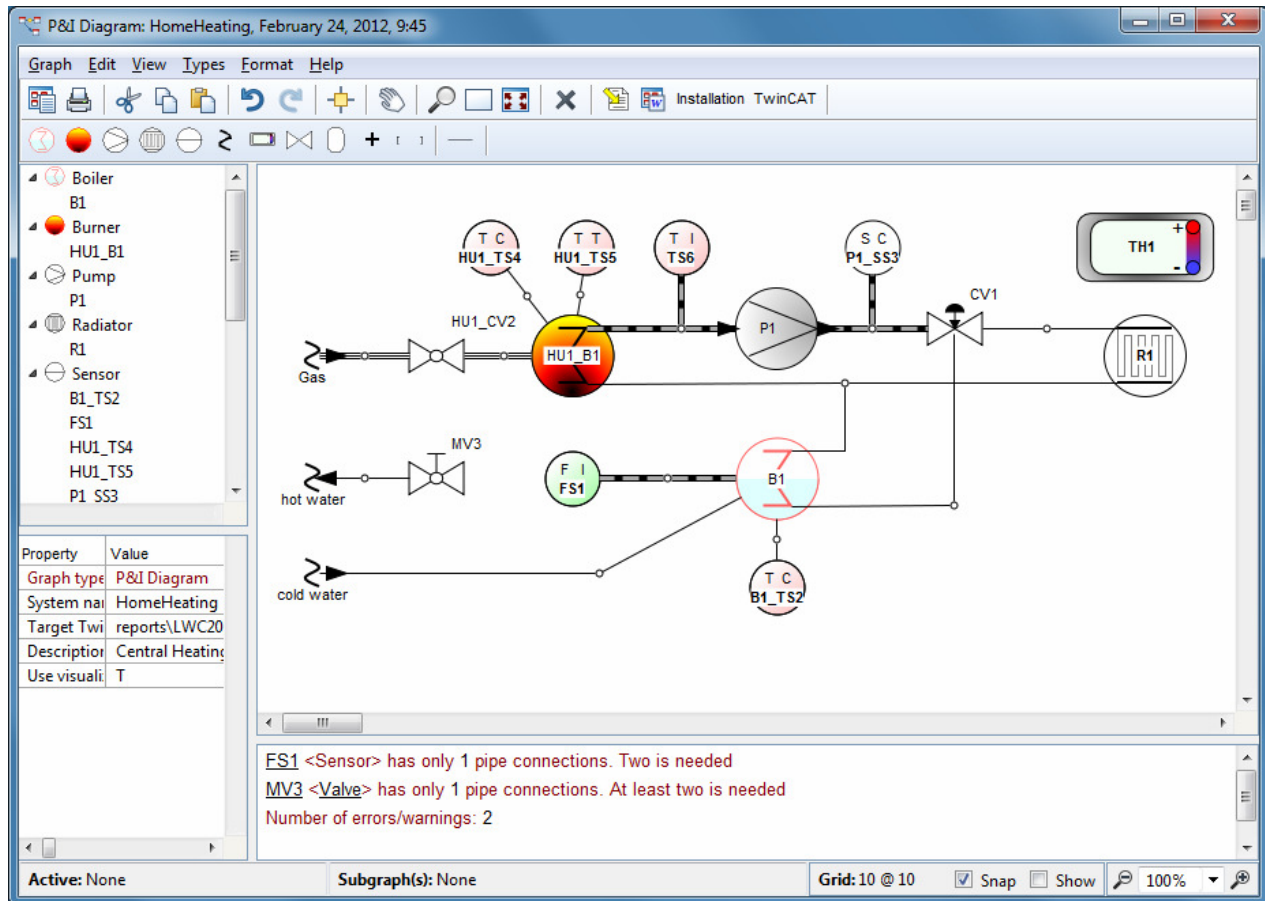


Diagram Editor shows the description of the LWC2012 heating system using the developed DSL. Second toolbar shows the main concepts of the DSL. Creation of the model has been made by choosing the domain concept from the toolbar and then adding it to the canvas. During model creation and editing Diagram Editor checks that the DSL definition is followed: either by checking the definition of the metamodel or by running the checking scripts written in MERL.

### 3 Define the P&I network in such a way that it is intuitive to the domain expert

While the notation shown above follows the well known and widely used notation of Piping and Instrumentation, it has been enhanced with optional coloring and error annotation. Figure below shows the same LWC2012 heating application using these visualization enhancements. These visualization options can be set on or off from the graphs property 'Use visualization'.

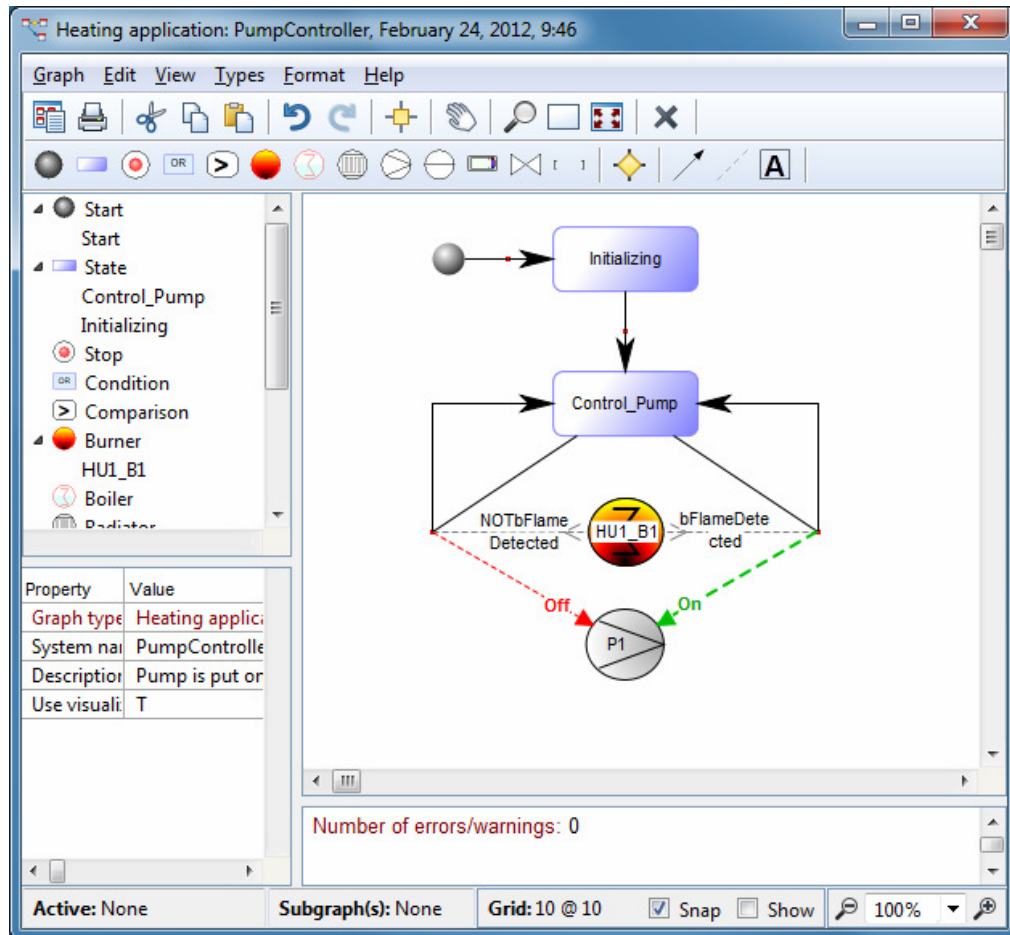


The editor also shows possible errors and incompleteness information as defined by the language engineer. Here two errors/warnings are reported: Sensor has only one connection and Valve has connection to one pipe only. By clicking the elements in the error report MetaEdit+ traces and selects the elements in the diagram to be updated.

While the P&I network is shown graphically it can also be presented textually for those who prefer textual specifications. Textual specifications can be generated (**Graph | Generators...**) from the diagram and the properties of the individual pipes and instruments can be accessed for editing from the text.

#### 4 Define the control behavior of the central heating system

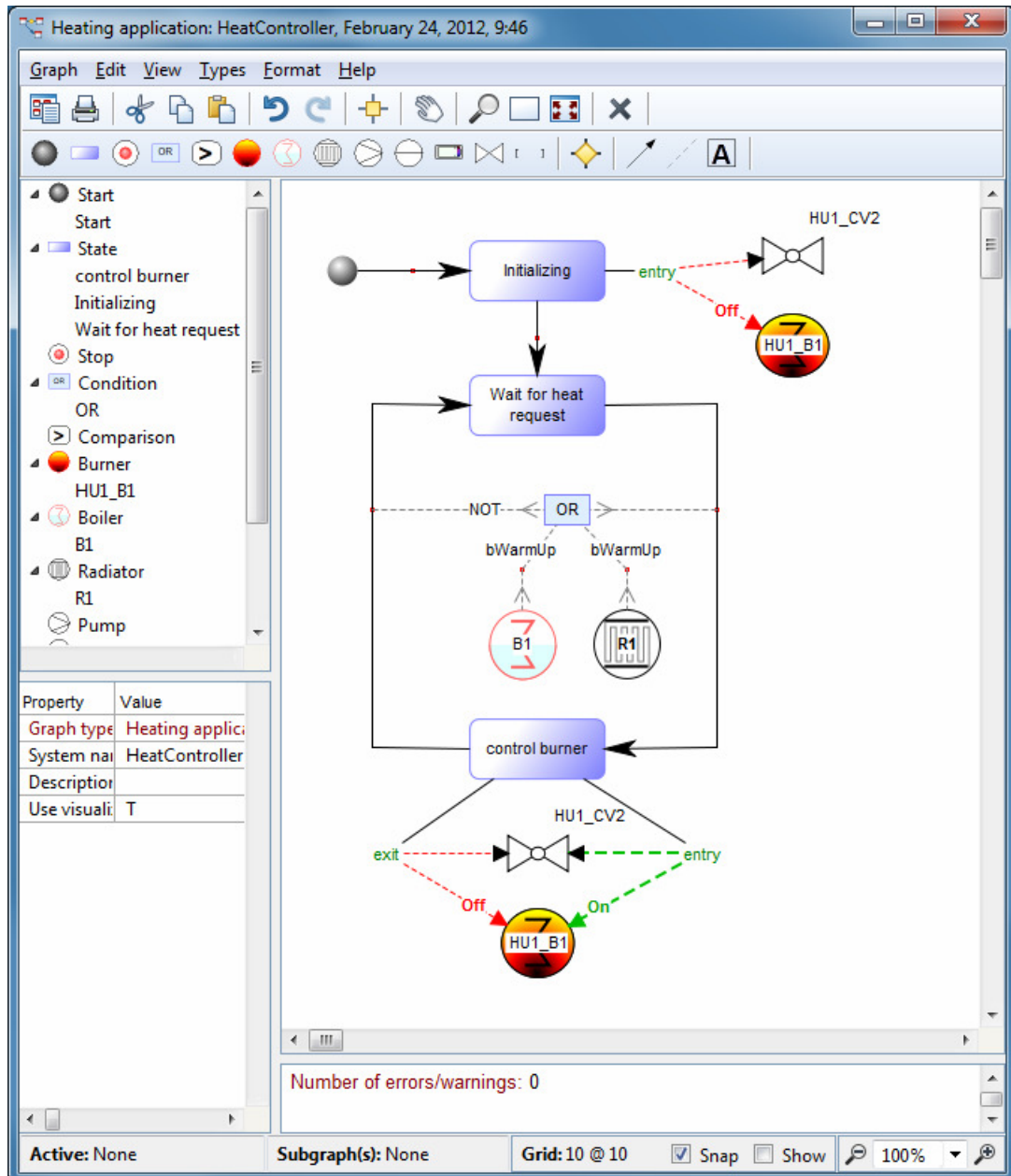
Behavior is defined with another graph type: 'Heating application'. The behavior description is defined for each relevant instrument of the P&I Diagram. Diagram Editor below shows the behavior of the 'Pump' P1. The language is based on a state machine, but is domain-specific by accessing conditions from the instruments only (like 'Boiler's' flame detection) as well as by being able to set actions for certain instruments only (like put 'Pump' on or off).



The behavioral model refers directly to the instruments defined in the P&I Diagram. In other words, an instrument is the same in both kinds of diagrams – no need to update a change in multiple places or keep string values matching. Diagram Editor (as well as other editors and browsers) allows tracing between these two kinds of diagrams as well as see how instruments are (re)used. By selecting **Info...** from the elements pop-up menu you can see the possible reuse cases. The languages support also checking consistency of the different diagrams. For example, if behavioral model uses instruments that are not defined in structural piping and instrumentation model they are reported as errors/warnings in the P&I Diagram.

For more complex conditions and comparisons the language has own constructs which available from the toolbar similarly to all other language constructs. Below a diagram for Heat Controller shows a constraint based on 'Radiator' *R1* and 'Boiler' *B1* warming. The example shows also entry and exit actions. While they are not necessarily, as the same functionality could be specified via state transitions too, they were added to the heating application language to mirror closely the reference implementation.



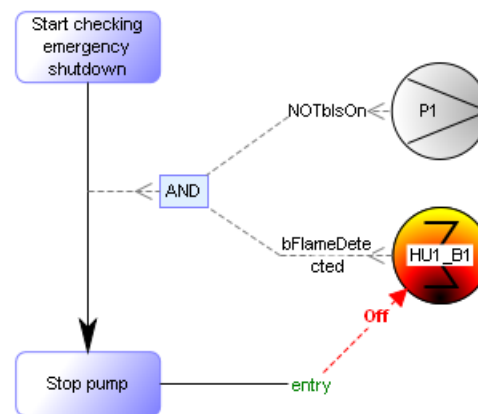


## 5 Add interlock/constraint definitions the define invariants of the central heating system.

Usually the best place to define invariants is in the definition of a modeling language itself or in the generator. This way language user always follows the constraints - even without knowing about them. Therefore the interlock/constraint definition can be supported similarly as done in task 1 ("be able to define the DSL elements").

If there is a need to model the interlocks/constraints then one solution would be defining them with the DSL too. That DSL would be still integrated with the other DSLs. For example, in the assignment of LWC2012, an example of invariant given is that if the burner is on then the pump must be running. Below such rule is defined with emergency shutdown action: if pump is not on and the burner is running then burner is put off. Other constraints could be added to the language too, but here already available DSL constructs can be used to describe invariants/constraints.





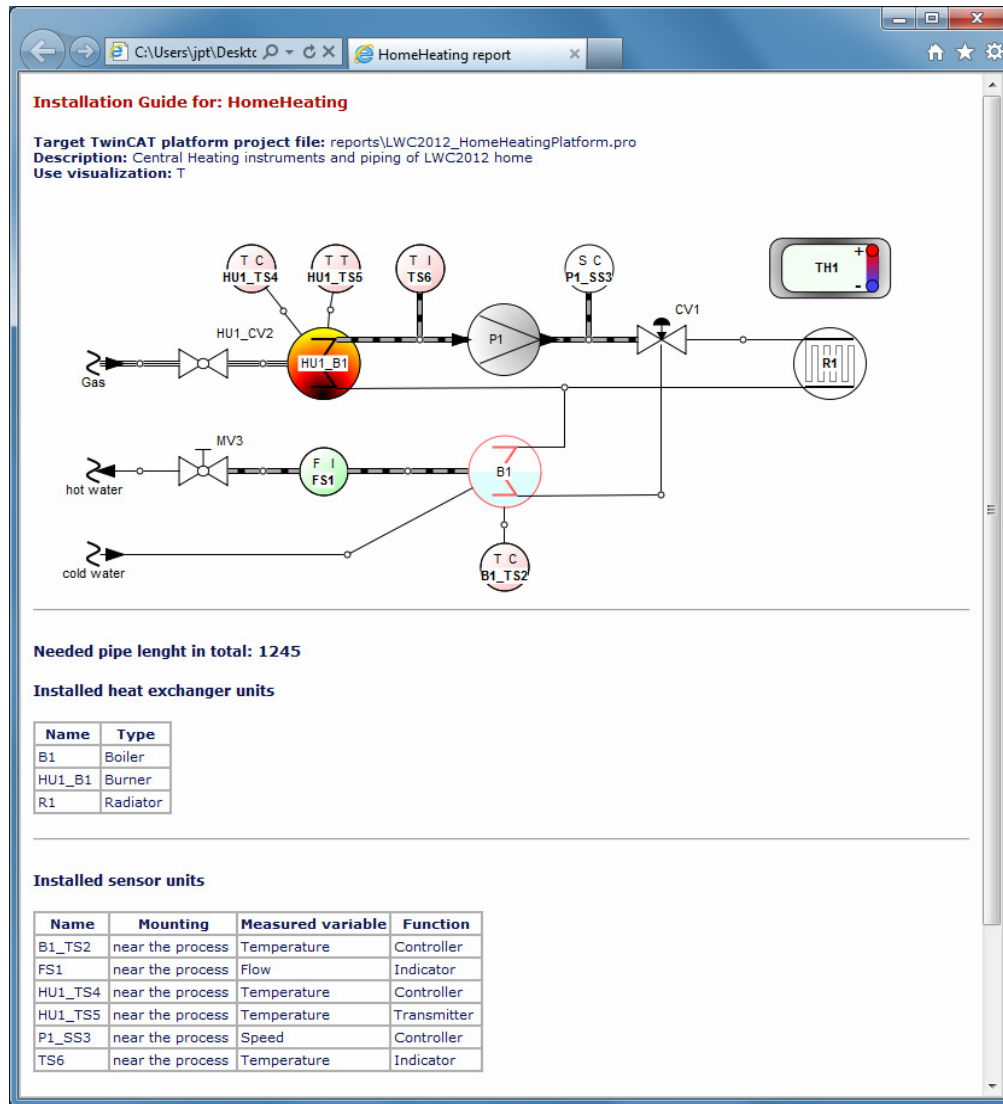
## 6 Generate structural definitions and stubs for a target

In the assignment of LWC2012, structural definitions include for example the data types of component types as well as various controller states. These are generated as a part of whole code generation process available from the toolbar of P&I Diagram. A sample of the generated data type code for radiator controller is shown below.

```

TYPE E_MDL_RadiatorController_SM_States :
(* RadiatorController states generated from MetaEdit+ *)
(
  (**** Initial States ****)
  MDL_RadiatorController_SM_Initial,
  (**** Normal States ****)
  MDL_RadiatorController_SM_INITIALIZING,
  MDL_RadiatorController_SM_MONITOR_ROOM_TEMPERATURE
);
END_TYPE
  
```

In addition to the code, other structural definitions are generated too. Figure below shows the generated installation guide describing the instruments needed as well as counting how much pipe is needed.



## 7 Generate control code for a target

All the code needed for the heating application, including those providing structural definitions as well as those generating function blocks with behavioral code are generated via a single main generator. In other words, the generator for the whole heating system calls the generators for structural data, logic, simulation, tasks etc.

To apply the generators choose **Graph | Generate...** and select 'TwinCAT-autobuild' or press 'TwinCat' button in toolbar of the Diagram Editor for P&I Diagram. The code generated with TwinCAT-autobuild option uses the platform of the homeheating system. This platform is similar to the .pro file of the assignment, except that it includes only the building blocks as the rest is generated. During the generation TwinCAT is started, code is imported and compiled for execution. If you just want to see the generated code, select generator 'expFiles'.

A sample of the generated code from the Heat Application behavior is shown below. The code is generated to mimic closely the approach used in the reference implementation. It is generated from 'Heating application' diagram *HeatController* (see task 4 above).

```
ACTION MDL_HeatController_SM_rg:

(* HeatController state machine generated from MetaEdit+ *)

MDL_HeatController_SM_bIsEntry := MDL_HeatController_SM_bATransitionWasPerformed;

IF MDL_HeatController_SM_bIsEntry THEN
```

```

        MDL_HeatController_SM_eLastState := MDL_HeatController_SM_eCurrentState;
        MDL_HeatController_SM_bATransitionWasPerformed := FALSE;
    END_IF

CASE MDL_HeatController_SM_eCurrentState OF

    MDL_HeatController_SM_Initial:

        IF NOT MDL_HeatController_SM_bATransitionWasPerformed THEN
            MDL_HeatController_SM_eCurrentState :=
                MDL_HeatController_SM_INITIALIZING;
            MDL_HeatController_SM_bATransitionWasPerformed := TRUE;
        END_IF

    MDL_HeatController_SM_CONTROL_BURNER:

        IF MDL_HeatController_SM_bIsEntry THEN
            MDL_HU_B1.On(); MDL_HU_CV2.Open();
        END_IF

        IF NOT MDL_HeatController_SM_bATransitionWasPerformed THEN
            IF NOT (MDL_R1.bWarmUp OR MDL_B1.bWarmUp) THEN
                MDL_HeatController_SM_eCurrentState :=
                    MDL_HeatController_SM_WAIT_FOR_HEAT_REQUEST;
                MDL_HeatController_SM_bATransitionWasPerformed := TRUE;
            END_IF
        END_IF

        IF MDL_HeatController_SM_bATransitionWasPerformed THEN
            MDL_HU_B1.Off(); MDL_HU_CV2.Close();
        END_IF

    MDL_HeatController_SM_INITIALIZING:

        IF MDL_HeatController_SM_bIsEntry THEN
            MDL_HU_B1.Off(); MDL_HU_CV2.Close();
        END_IF

        IF NOT MDL_HeatController_SM_bATransitionWasPerformed THEN
            MDL_HeatController_SM_eCurrentState :=
                MDL_HeatController_SM_WAIT_FOR_HEAT_REQUEST;
            MDL_HeatController_SM_bATransitionWasPerformed := TRUE;
        END_IF

    MDL_HeatController_SM_WAIT_FOR_HEAT_REQUEST:

        IF NOT MDL_HeatController_SM_bATransitionWasPerformed THEN
            IF MDL_R1.bWarmUp OR MDL_B1.bWarmUp THEN
                MDL_HeatController_SM_eCurrentState :=
                    MDL_HeatController_SM_CONTROL_BURNER;
                MDL_HeatController_SM_bATransitionWasPerformed := TRUE;
            END_IF
        END_IF

    ELSE
        F_Assert(FALSE, 'Wrong MDL_HeatController_SM_rg state identifier');
    END_CASE

END_ACTION

```

## 8 Generate a mock representation (simulation) of the defined central heating system that behaves like it and that can be used to test the other generated stuff against

'TwinCAT-autobuild' and 'expFiles' generators produce also the simulation for TwinCAT. In TwinCAT System Manager, after importing the created project info (.tpy), you must link the inputs and outputs of the produced application into those used for simulation and generate the mappings in TwinCAT System Manager. After that you can run the generated simulation.

## 9 Generate operations interface for higher level software that enables the use of the generated functionality through that API

Generation of interface code is done similarly to any other generator of MetaEdit+: writing it with MERL accessing the same heating application models. An example of such interface code generation is made for accessing data and transmitting values from the sensors.

A sample of the generated API code is shown below. Here the API includes only those sensors that are available for the given heating system (see P&I Diagram in task 3). You can run the generator from Editor for P&I Diagrams by selecting **Graph | Generate...** and then selecting 'Sensor Interface API'.

Sensors of HomeHeating can be accessed with the following API:

```
long getTemperature (string SensorID);
(return the current temperature data from the named sensor)

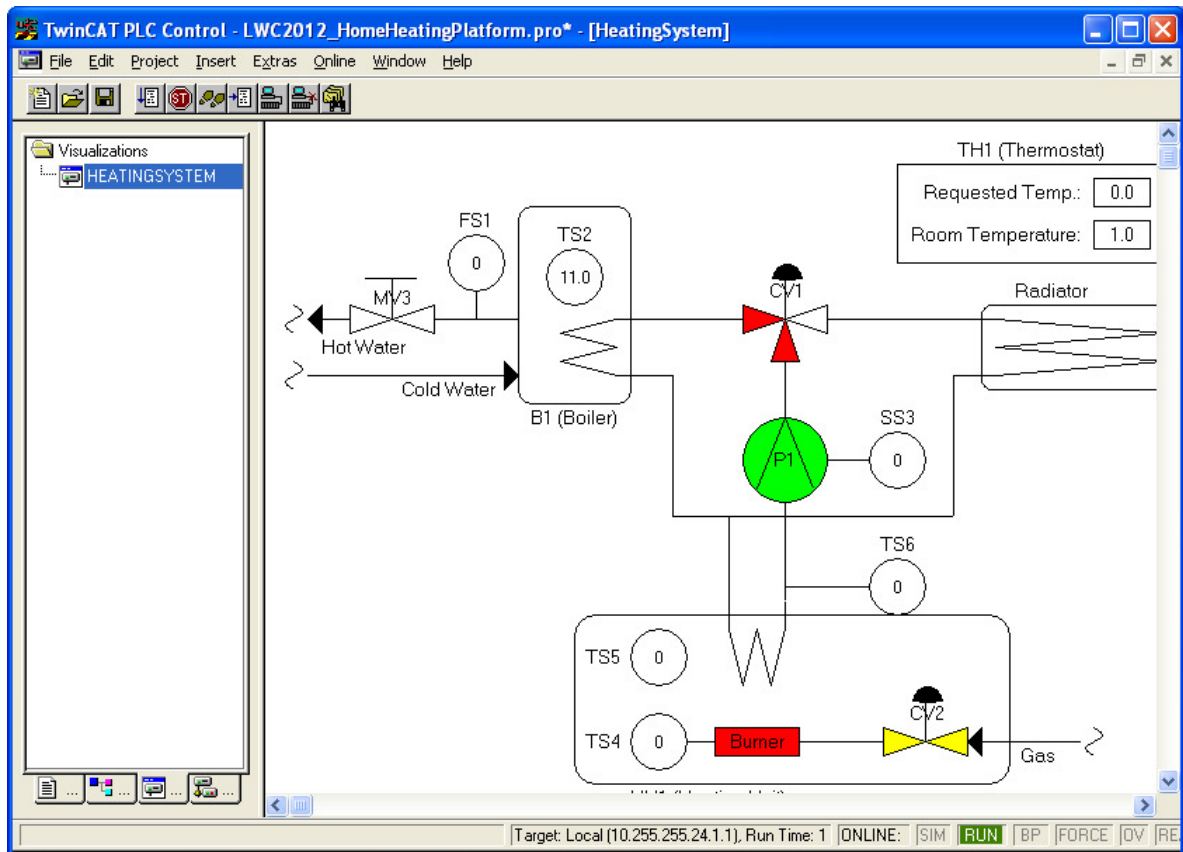
long getFlow (string SensorID);
(return the current flow data from the named sensor)

long getSpeed (string SensorID);
(return the current speed data from the named sensor)

void transmitTemperature (string SensorID);
(transmits the temperature from the named sensor)
```

## 10 Generate visualization software that can connect to the target to visualize the dynamic behavior

TwinCAT provides a possibility to visualize the running system. This approach can be applied with the code generated with MetaEdit+ as well. The figure below shows a snapshot of the simulation for the generated heating application. Here the visualization elements are those available from TwinCAT.



An alternative approach, and often a better one, is to use MetaEdit+ directly to visualize the execution. In other words, MetaEdit+ animates P&I Diagram or visualizes the behavioral part of the system by animating state changes of the controllers. The advantage of this approach is that there is no need to create again models of piping and instrumentation (as above) or of behavior for the simulation purposes. Most importantly, this way MetaEdit+ provides a direct model-level debugging using the original models as visualizations: if something was found wrong during the simulation engineers can correct the simulated diagram and run the generators again.

To visualize dynamic behavior in MetaEdit+, the target system (TwinCAT or even the real target system depending on its capabilities) can call MetaEdit+ API. For example, with API's *animate* command the pipes, instruments, and states could be animated directly in Diagram Editor of MetaEdit+. MetaEdit+'s API is based on SOAP / Web Services / .NET standard for application integration that is available for almost any programming language and office tools (e.g. excel). The implementation of visualization support from TwinCAT is left to the reader as MetaEdit+ already provides capabilities for visualization. To use MetaEdit+ enabled visualization, the generated code would include object identifiers that MetaEdit+ uses for model elements. Therefore, the generator developer defines which model elements should be visualized: instruments as in LWC2012 reference implementation or also states, actions etc. defined in the behavioral models. MetaEdit+ API is not restricted to animation only as it can be used also to modify, create and delete the elements in models as well as the representation of the model elements.

---

## Conclusions

MetaEdit+ is a mature language workbench for developing and using domain-specific modeling languages and generators. We demonstrated how the tasks of Language Workbench Challenge 2012 were implemented in MetaEdit+. You can investigate the languages and generators - as well as extend them - by downloading MetaEdit+ and the related repository (see section 'Getting started' in the beginning for details).

For more information about MetaEdit+, including tutorials, case studies, pricing, visit <http://www.metacase.com>.



---

## References

- [1] MetaEdit+ User Guides, <http://www.metacase.com/support/45/manuals/>
- [2] Graphical Metamodeling,  
<http://www.metacase.com/support/45/manuals/Graphical%20Metamodeling.pdf>