Version 5.0

Heating System Example

MetaCase Document No. PLC-5.0
Copyright © 2012 by MetaCase Oy. All rights reserved
First Printing, 2nd Edition, September 2012

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 14 641 000
Fax: +358 420 648 606
E-mail: info@metacase.com
WWW: http://www.metacase.com

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including but not limited to photocopying, without express written permission from MetaCase.

MetaEdit+ is a registered trademark of MetaCase. The other trademarked and registered trademarked terms, product and corporate names appearing in this manual are the property of their respective owners.

Preface

The heating system example illustrates how heating applications can be modeled and generated based on Domain-Specific Modeling (DSM). To achieve this, two integrated domain-specific modeling languages are implemented into MetaEdit+ along with a generator for producing PLC code. Using these modeling languages, a developer can design heating applications using directly the domain concepts of the domain, like pipes, pumps, valves, pressure sensors and their related behavior like opening or closing valves. Generators are used to produce the executable code, integrated into a PLC software development environment (TwinCAT). Generators are also used for producing installation guidelines, documentation, and model checking.

The rest of this document describes the language and generators for the developing heating systems as well as how they were implemented with MetaEdit+. First, we inspect shortly the modeling languages with some examples and then we discuss selected topics on modeling language and generator specification. These topics are set by the LWC2012 challenge (http://languageworkbenches.net).

For exploring the heating system example thoroughly, the following things are required:

- MetaEdit+ for trying out the languages and generators. The PLC heating system example can be found from the demo repository, from the project named 'Heating system'. For further information about MetaEdit+, please refer to the MetaEdit+ User's Guide[1].
- Beckhoff TwinCAT software system for running the generated application. TwinCAT can download from http://www.beckhoff.de/english.asp?twincat/default.htm. Download TwinCAT 2.11 R2, Build 2038. There may also be newer versions available.

We expect that you have knowledge about using MetaEdit+. If you want to extend the DSM further − add notational symbols, additional constraints, generators or by modifying dialogs and toolbars for the modeling tool − you should have MetaEdit+ Workbench or the evaluation version available from http://www.metacase.com.

[1] MetaEdit+ User Guides, http://www.metacase.com/support/50/manuals/

1 DSM for heating systems

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 'Heating system' project contains one P&I Diagram describing the piping and instrumentation and five heating applications which run in the controllers.

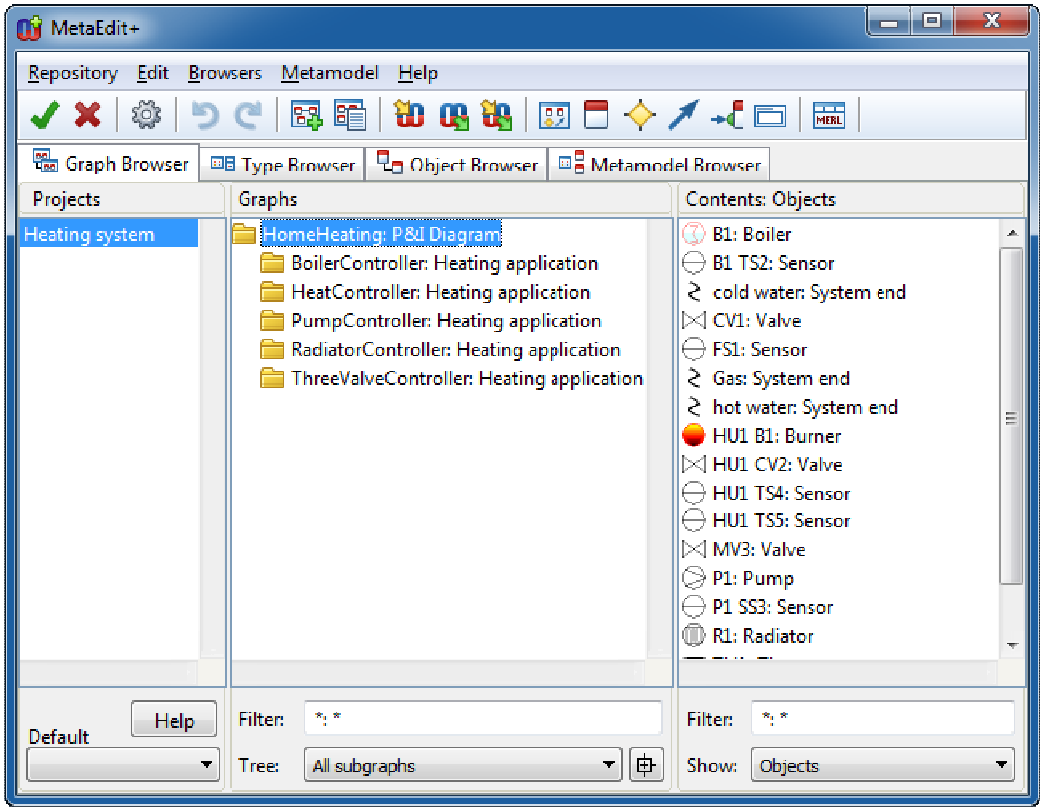


Figure 1-1. MetaEdit+ main window showing the contents of heating system project

1.1 SYSTEM STRUCTURE

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

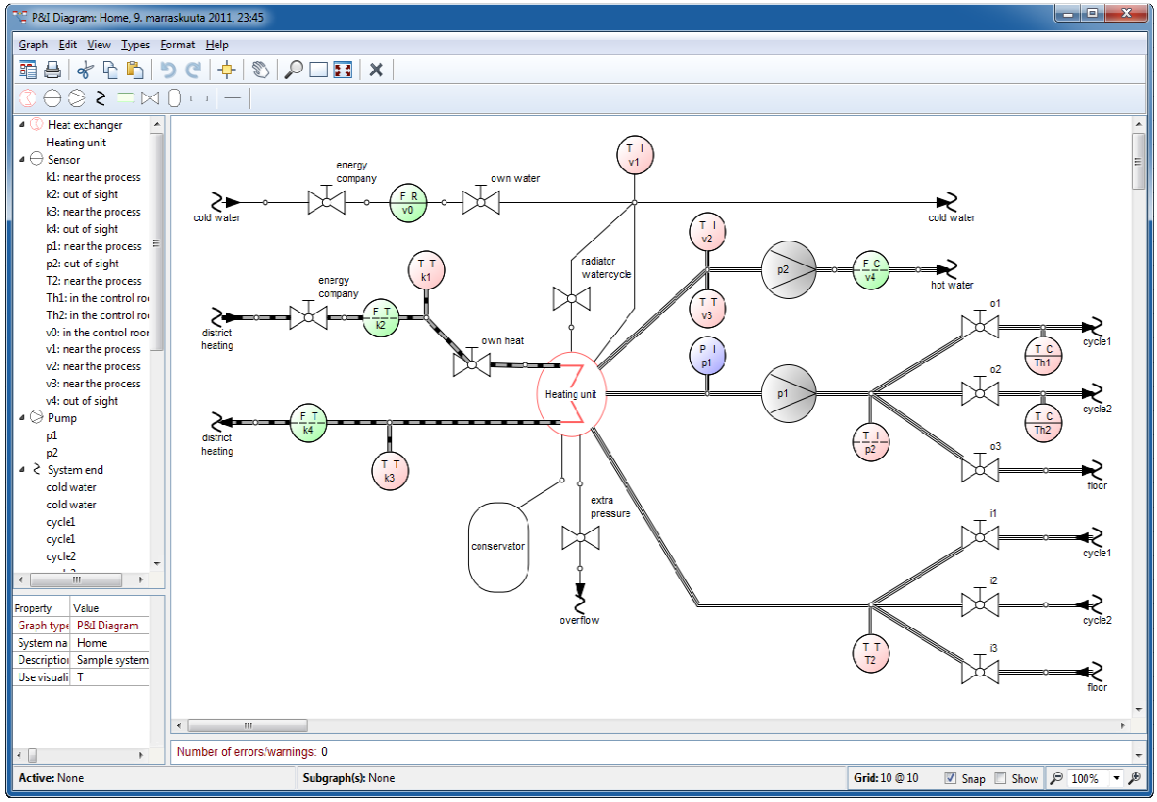


Figure 1-2. Structure: Pipes and Instrumentation

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 types of diagrams.

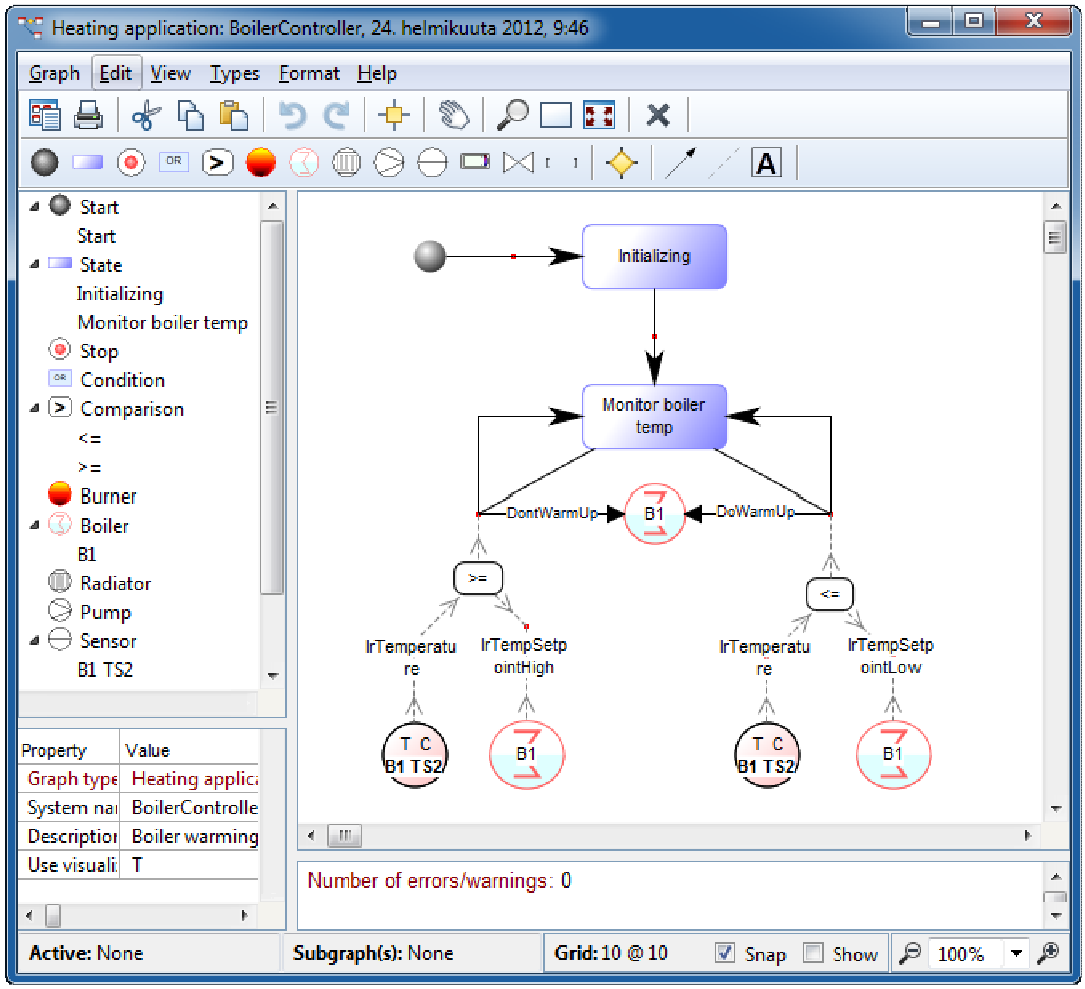


Figure 1-3. Behavior of boiler controller

1.3 GENERATORS

Generators are available for both languages. Most important generators are those for P&I Diagram:

- '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.

2 Implementing the DSM language for the heating system domain

This section describes how domain-specific modeling languages and generators are implemented in MetaEdit+.

2.1 DEFINING LANGUAGE ELEMENTS

Elements of the modeling language can be defined in MetaEdit+ with a graphical metamodeling language or with form-based metamodeling tools. We outline here these tools used to define languages for heating systems. For a complete description of these tools, see MetaEdit+ Workbench User's Guide[2].

2.1.1 Graphical metamodeling

Language elements, abstract syntax part of the language, can be defined with a graphical modeling language. Figure below shows the partial metamodel of a language used to specify control behavior of the central heating system: state transitions of the 'Heating application' modeling language. A blue rounded rectangle symbol indicates an object type or a set of object types, an orange diamond symbol shows the 'transition' relationship type and a green circle is used to describe the role types.

[2] MetaEdit+ Workbench Guide, http://www.metacase.com/support/50/manuals/

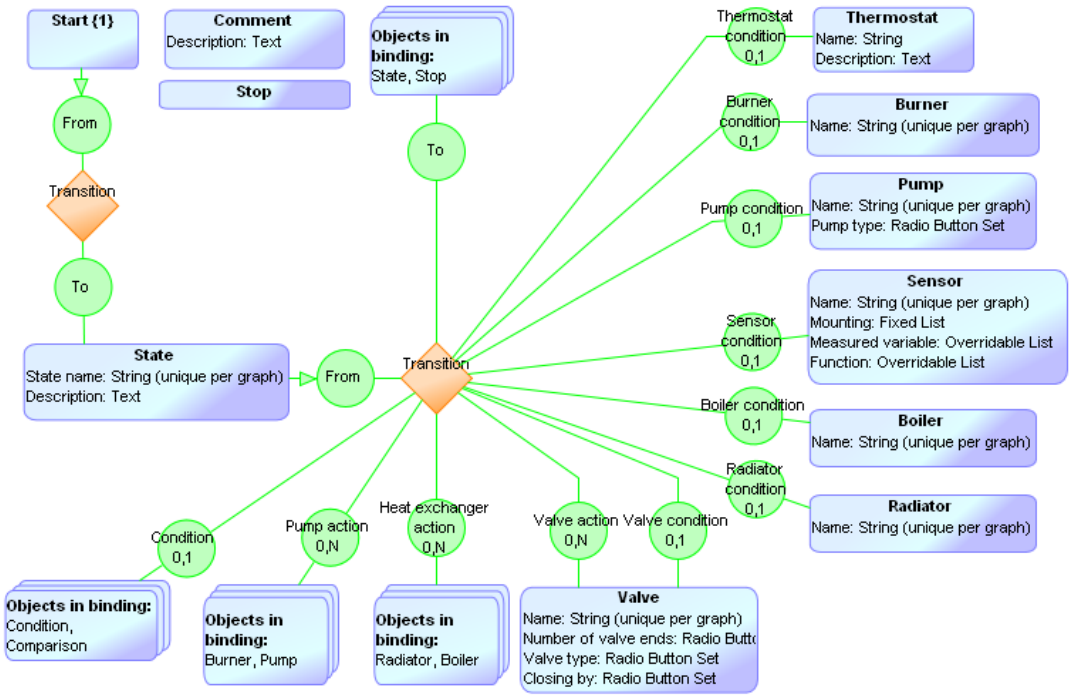


Figure 2-1. Metamodel of heating application language (partial)

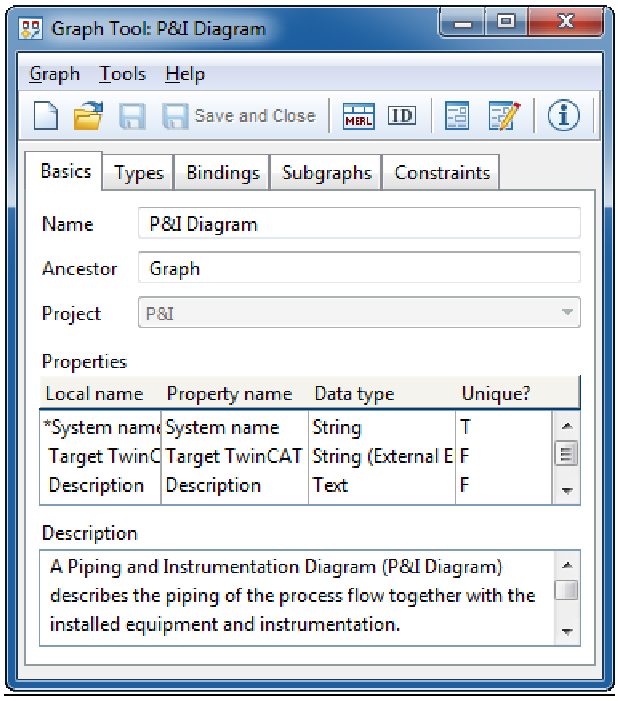
After the metamodel, as above, is drawn in Diagram Editor of MetaEdit+ it can be instantiated and tested immediately with the modeling editors providing full editor functionality (copy/paste (special…), undo, replace, trace, print etc.). Once the language definition is complete it can be given for the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

2.1.2 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.

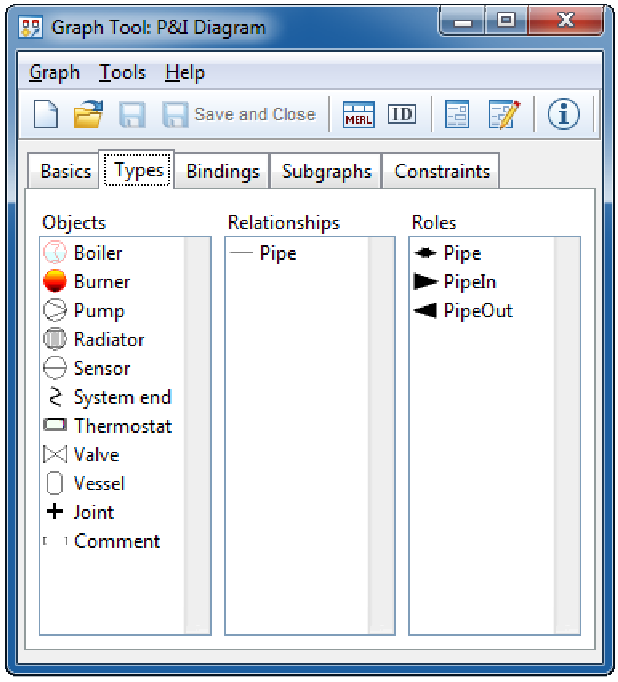
Most importantly, the definition of language elements is automatically applied in the various modeling editors (Diagram, Matrix, Table), browsers (Graph, Object, Type) and generators. This supports agile and incremental language definition: updates to the modeling language can be tested immediately and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

Language concepts (abstract syntax)

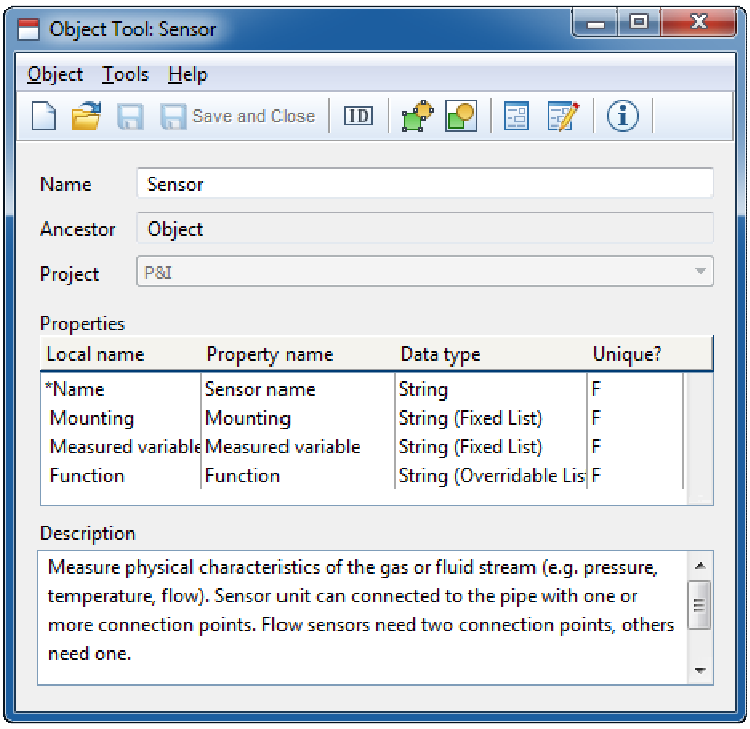


Graph tool defines the individual languages. Here a definition for a 'P&I Diagram' is given. A diagram itself has properties, and each property has a more detailed definition. For example, 'System name' is an identifying (marked with '*') property. It is of string data type and its value must be unique: 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…**).

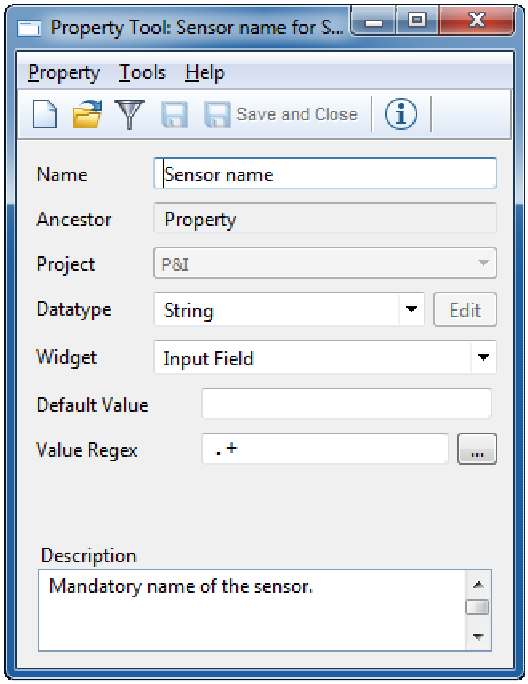


Types tab in the Graph Tool shows the individual language concepts: object types, relationship types and role types. These different type of concepts can be here added or removed from the language.

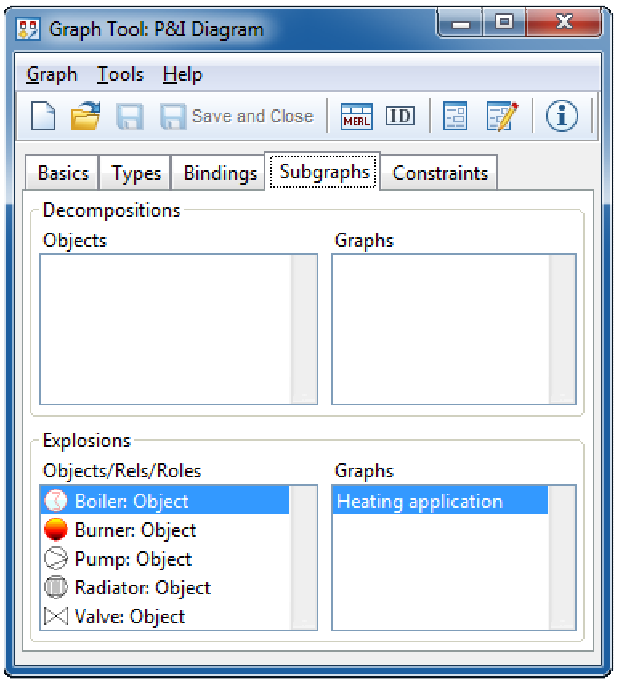


For each type, a form-based metamodeling tool shows its definition. Here Object Tool shows the definition of a 'Sensor': its name, ancestor type, four different property types and documenting description.

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



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 heating systems: 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.

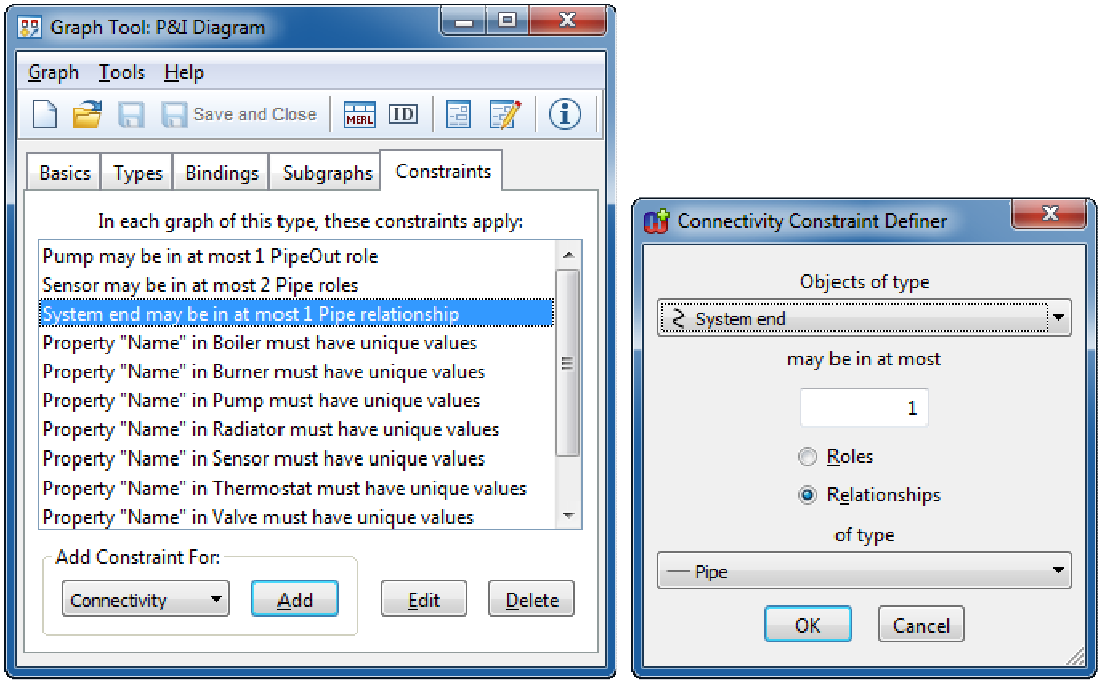


Figure 2-2. Defining constraints of the language

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.

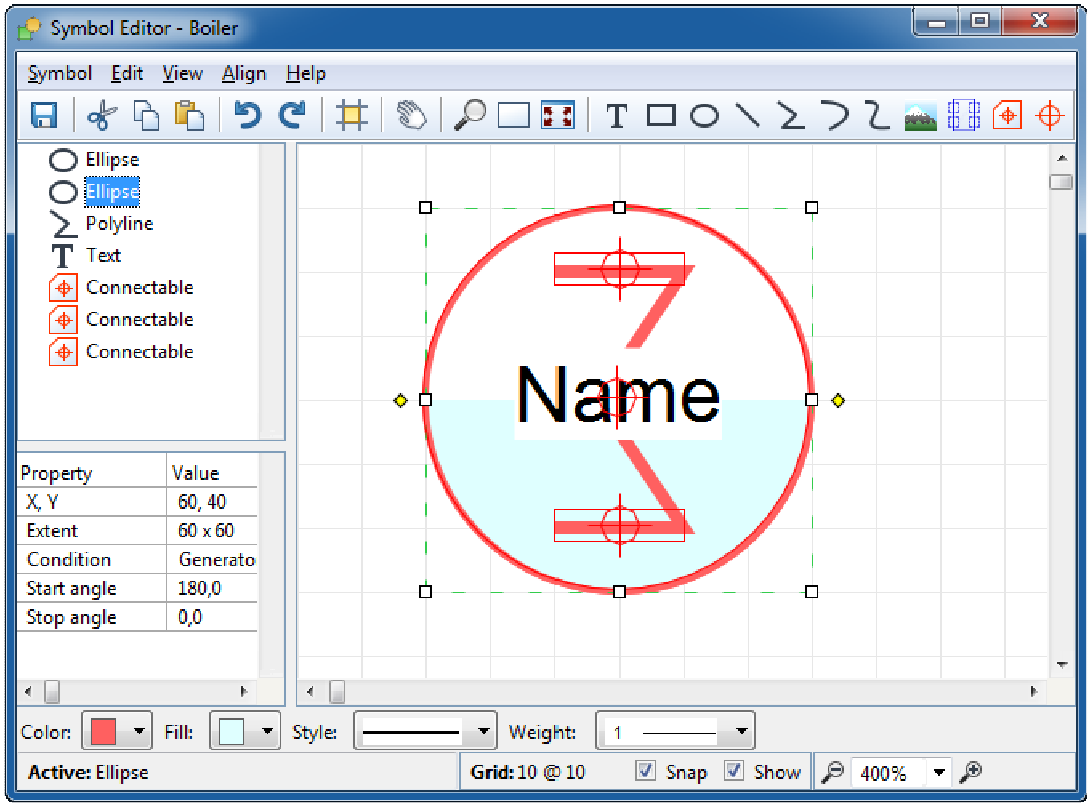


Figure 2-3. Symbol definition for the Boiler

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 heating systems, 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.

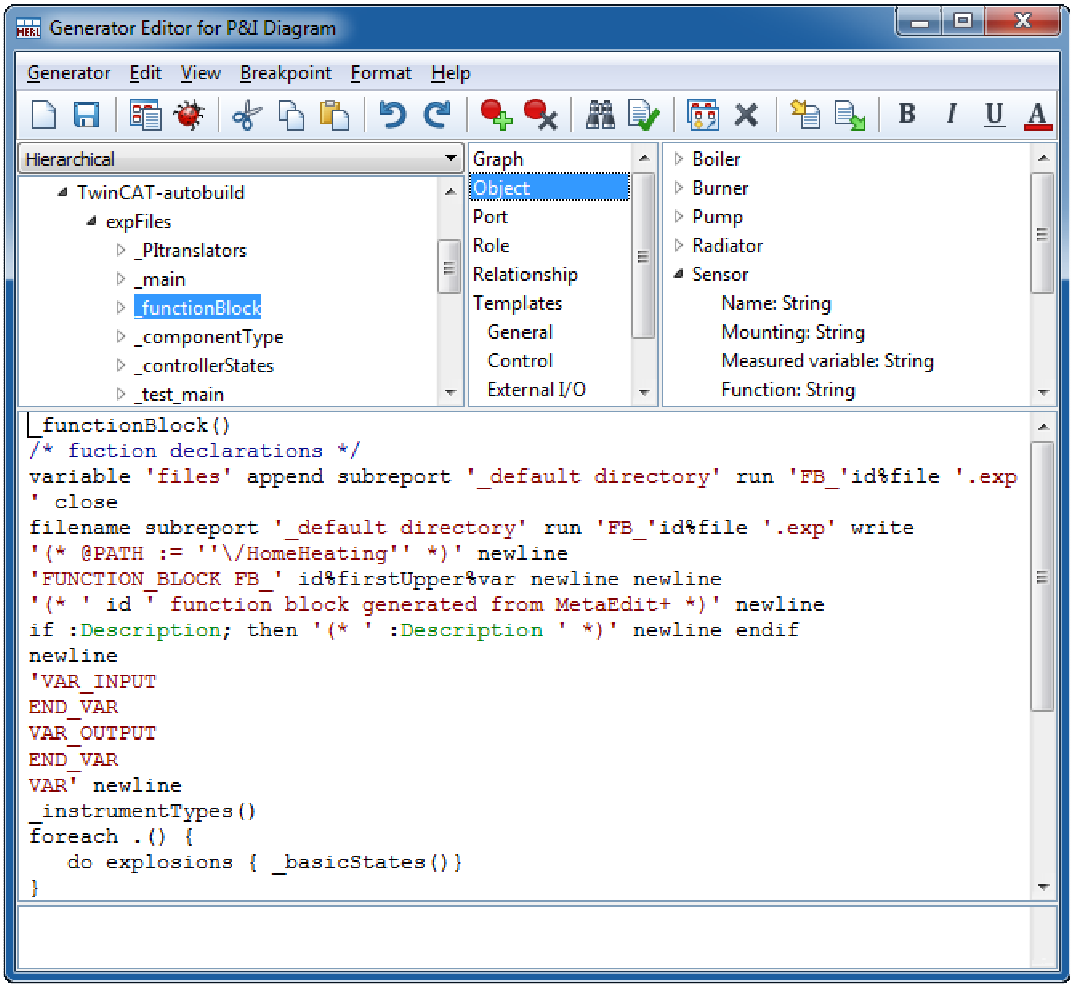


Figure 2-4. Generator definition in MERL

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.

2.2 DEFINING A P&I NETWORK OF THE HEATING SYSTEM

At any point of time during the language 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 DSM support.

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.

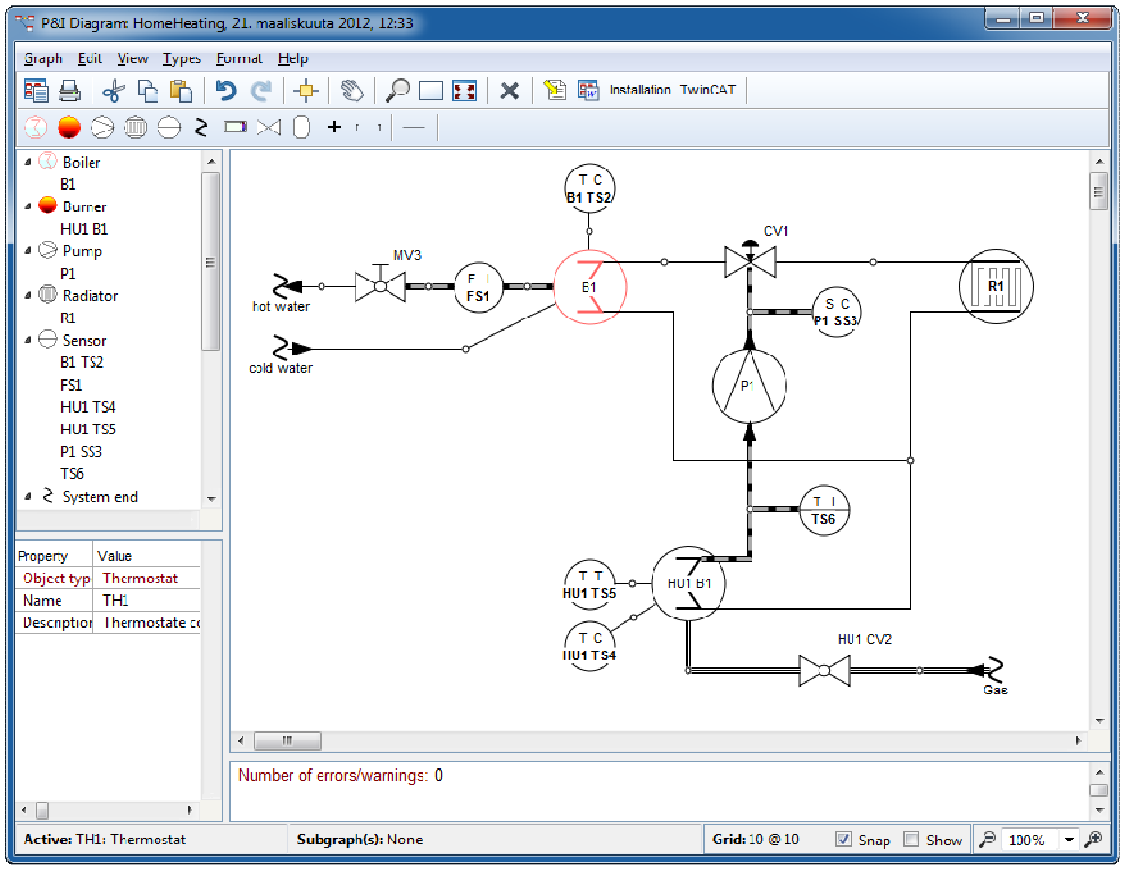


Figure 2-5. Instruments and their pipe connections

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.

2.3 IMPROVED VISUALIZATION OF THE P&I NETWORK

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 heating application using these visualization enhancements. These visualization options can be set on or off from the graphs property 'Use visualization'.

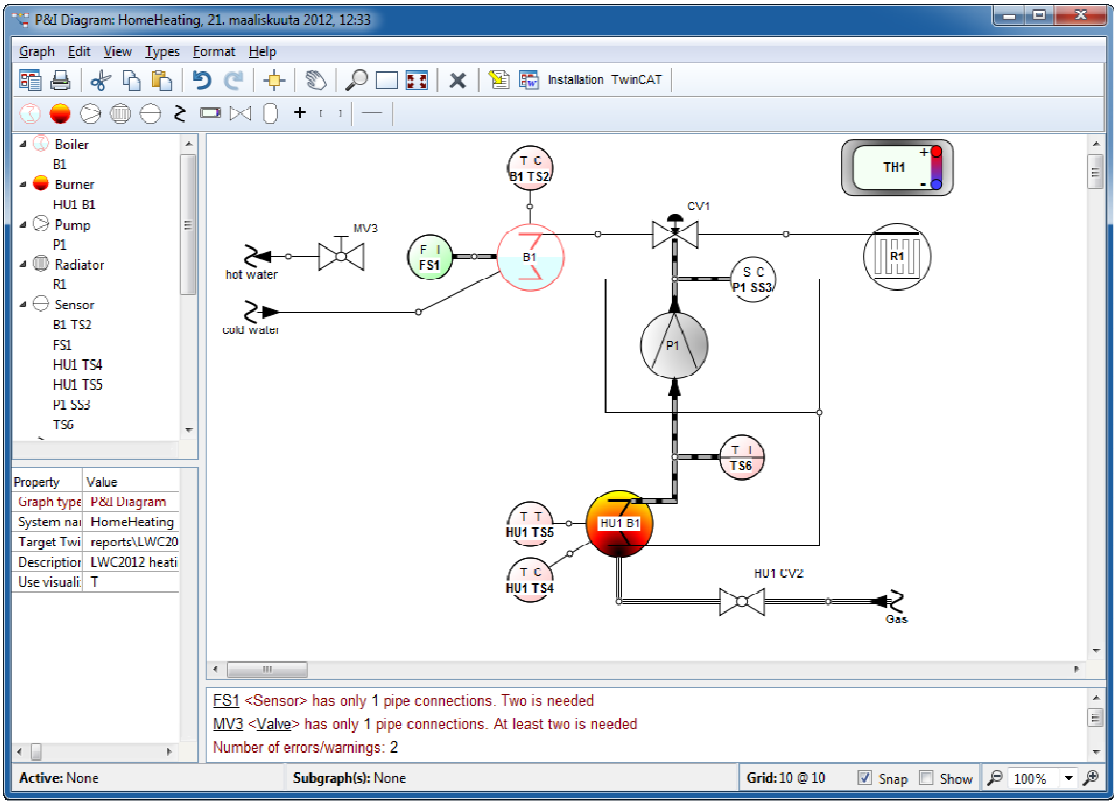


Figure 2-6. Visualizing nstruments in P&I Diagram

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.

2.4 CONTROL BEHAVIOR OF THE 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).

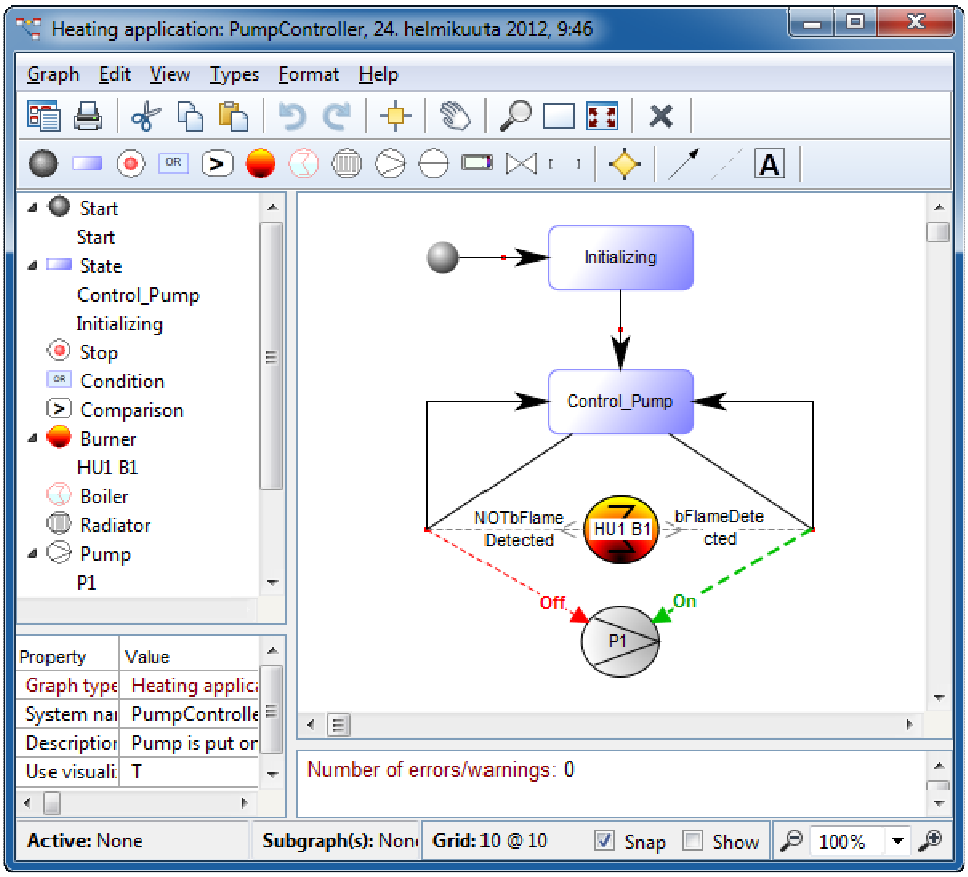


Figure 2-7. Behavior of pump controller

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.

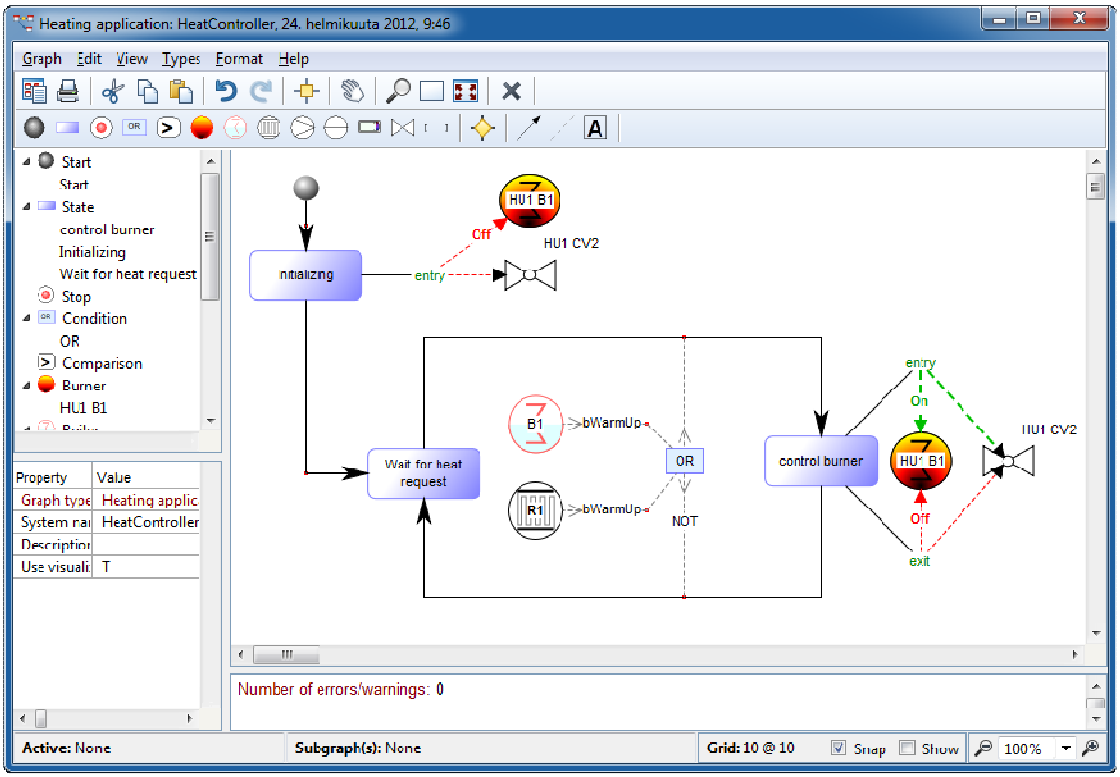


Figure 2-8. Behavior of heating controller

2.5 SPECIFYING INTERLOCK/CONSTRAINT DEFINITIONS 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 Section 2.1, defining language 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. An example of invariant 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.

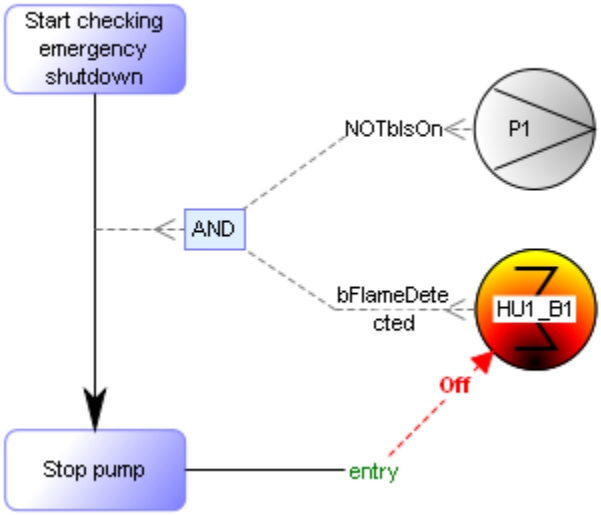


Figure 2-9. Modeling invariant/interlocks with the DSL

2.6 GENERATING STRUCTURAL DEFINITIONS AND STUBS FOR A TARGET

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.

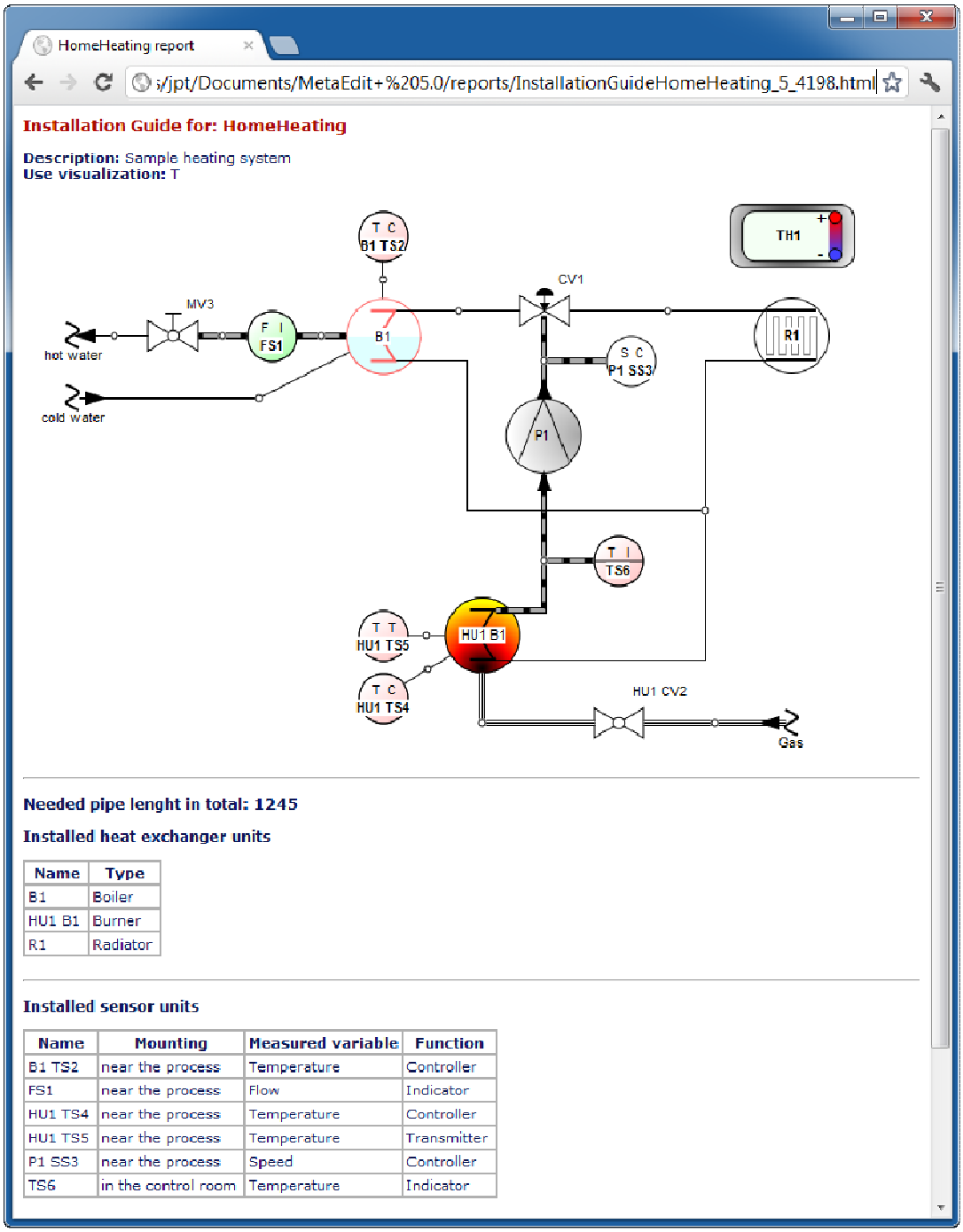


Figure 2-10. Generated installation documentation

2.7 GENERATING 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 includes the basic building blocks which the code generated from the models uses.

In this example the generator also produces the platform: it is a part of the generator (one subreport in the 'TwinCAT-autobuild' generator). In practice the platform would be a separate project file in TwinCAT. During the autobuild generation TwinCAT is started, code is imported and compiled for execution. When creating the project TwinCAT asks to choose the target system. Select PC. 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 Section 2.4 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
```

2.8 GENERATING A MOCK REPRESENTATION (SIMULATION) OF THE DEFINED HEATING SYSTEM

'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.

2.9 GENERATING INTERFACE FOR HIGHER LEVEL SOFTWARE

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 Section 2.3). 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);
(returns the current temperature data from the named sensor)

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

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

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

2.10 VISUALIZING 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.

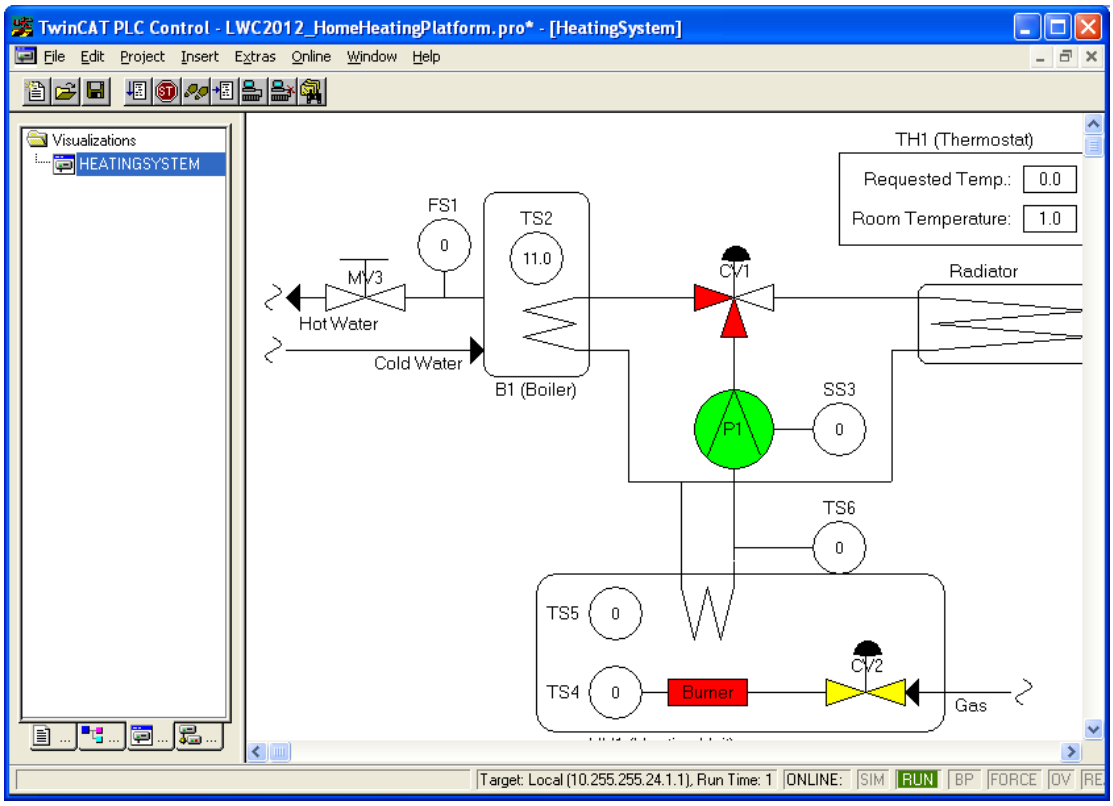


Figure 2-11. Visualizing the heating system in TwinCat

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 the state changes of the controllers. The advantage of this approach is that there is no need to create duplicate models of piping and instrumentation (as above) or of behavior for the simulation purposes. Most importantly, this way MetaEdit+ provides 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 the MetaEdit+ API. For example, with the 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 of the structural models or 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.

3 Conclusions

In this example, we have demonstrated a DSM for PLC heating system application development. With the domain-specific language we can model heating applications using two languages: one language targets the structure of the system specifying how instruments are connected with pipes, and another, yet integrated language, describes the behavior of the various controllers.

On the DSM definition side we focused on several a few areas of language design: integrated two languages, generating the code so that it integrated with an existing PLC development tool, and producing various other artifacts like installation guide, sensor API and model checking.

This DSM solution is implemented as any other modeling language and generator in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements of modeling or code generation. You could for example extend the framework, change the domain rules in the language, or produce other kind of code than the current state-based PLC code. The choice is yours because with DSM you control both the language as well as the generators.