Version 5.6

Heating System Example

MetaCase Document No. PLC-5.6

First Printing, 1st Edition, October 2025

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

E-mail: info@metacase.com
WWW: https://www.metacase.com

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 in MetaEdit+, along with a generator for producing PLC code. Using these modeling languages, a developer can design heating applications directly using the concepts of the domain, such as pipes, pumps, valves and pressure sensors, along with their related behavior such as opening and 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 developing heating systems, as well as how they were implemented with MetaEdit+. First, we briefly inspect the modeling languages with some examples, and then we discuss the modeling language and generator specification.

To explore 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 https://www.beckhoff.de/english.asp?twincat/default.htm. Download TwinCAT 2.11 R2, Build 2038, or other compatible systems.

We expect that you have basic knowledge about using MetaEdit+. If you want to extend the DSM solution 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 https://www.metacase.com.

[1] MetaEdit+ User Guides, https://www.metacase.com/support/

1 DSM for heating systems

The heating applications support in MetaEdit+ includes two integrated languages and a number of generators. In addition to editors, MetaEdit+ also provides 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 'Heating system' project has been opened and contains one P&I Diagram describing the piping and instrumentation, and five Heating applications which run in the controllers.

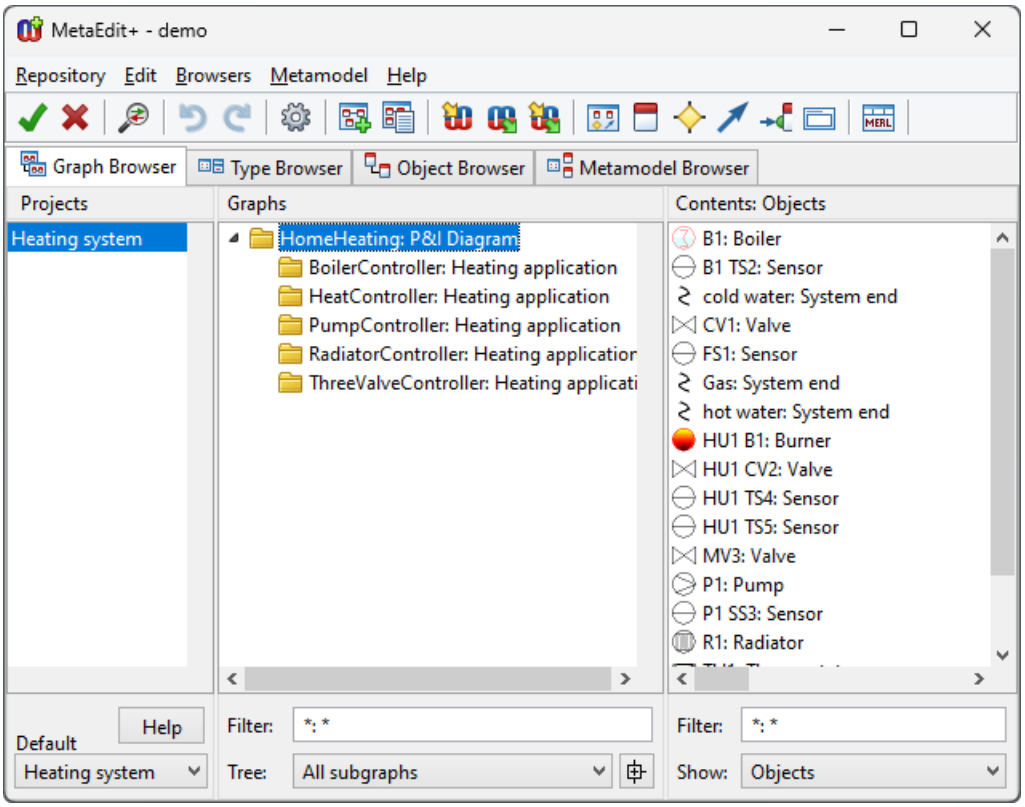


Figure 1-1. MetaEdit+ main window showing the contents of heating system project

1.1 SYSTEM STRUCTURE

The P&I Diagram specifies pipe connections among the various devices: pieces of equipment and instrumentation. The behavior of each controllable device can be specified in a subdiagram using another domain-specific language.

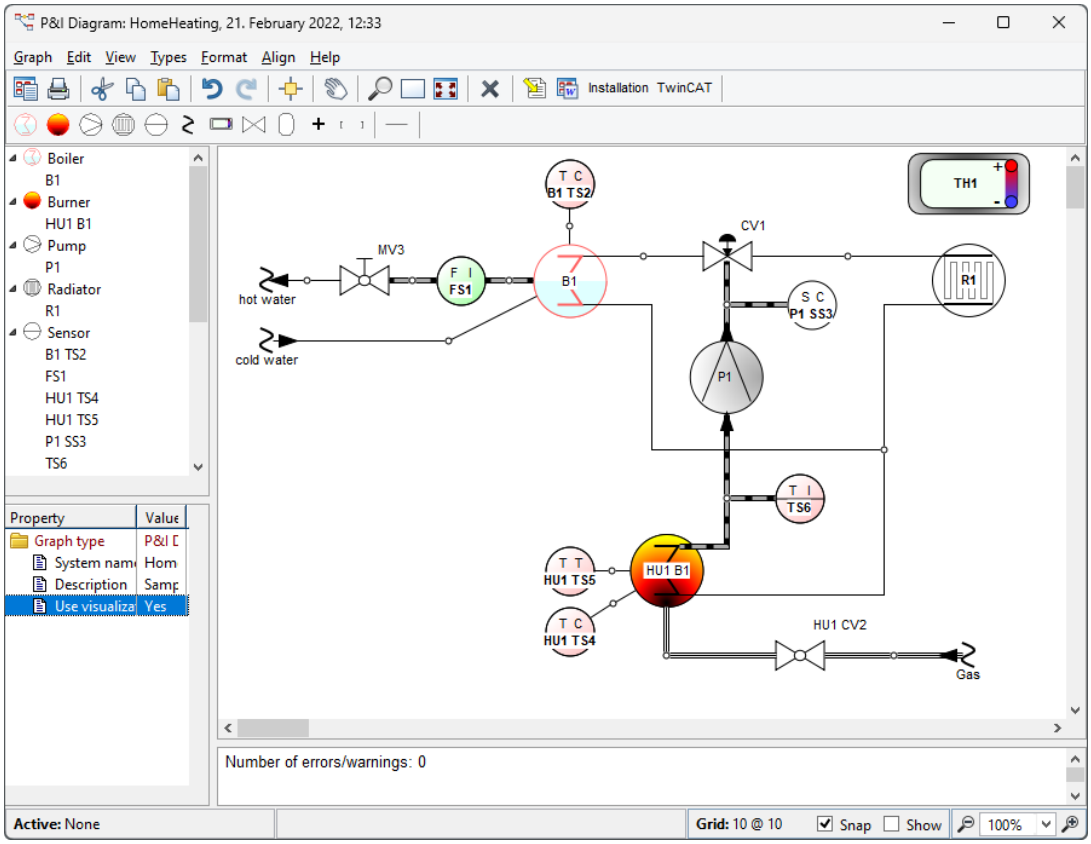


Figure 1-2. Structure: Pipes and Instrumentation

1.2 CONTROL BEHAVIOR

The Heating application language specifies the control logic of the system: the 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.

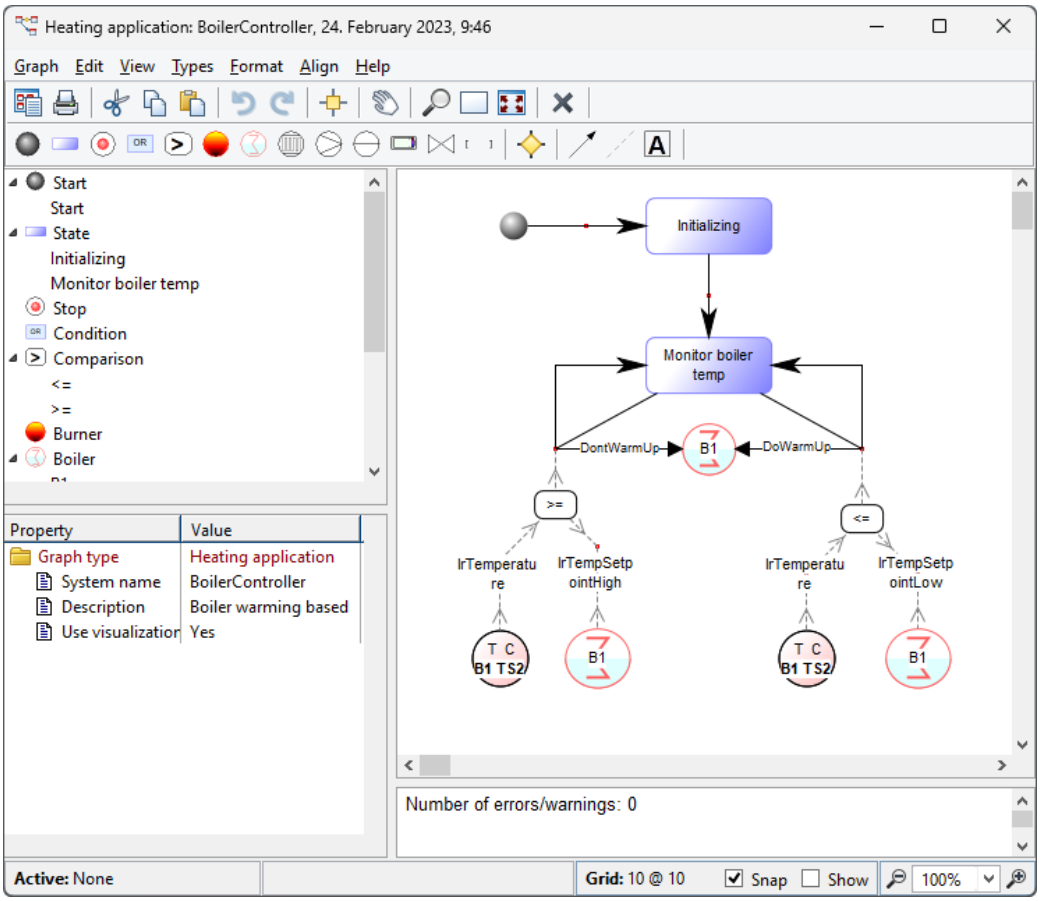


Figure 1-3. Behavior of boiler controller

1.3 GENERATORS

Generators are available for both languages. The most important generators are those for the P&I Diagram:

- 'expFiles' produces the code in the format of TwinCAT .exp files. The generated code provides the control logic blocks, its simulation and 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 a hardware 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 as a Word document.

In addition to these generators other generators are defined for model checking (shown at the bottom of Diagram Editor), generating interface descriptions, and producing a 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

The abstract syntax and rules of the modeling language can be defined in MetaEdit+ with a graphical metamodeling language or with form-based metamodeling tools. The concrete syntax is defined with a WYSIWYG vector graphics Symbol Editor. The semantics is defined as translational semantics with generators. We outline here those tools used to define the 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, the abstract syntax part of the language, can be defined with a graphical modeling language. The figure below shows the partial metamodel of a language used to specify control behavior of a central heating system: a state transition diagram language definition for Heating applications. A blue rounded rectangle symbol indicates an object type or a set of object types, an orange diamond symbol shows a relationship type and a green circle is used to describe the role types.

[2] MetaEdit+ Workbench Guide, https://www.metacase.com/support/

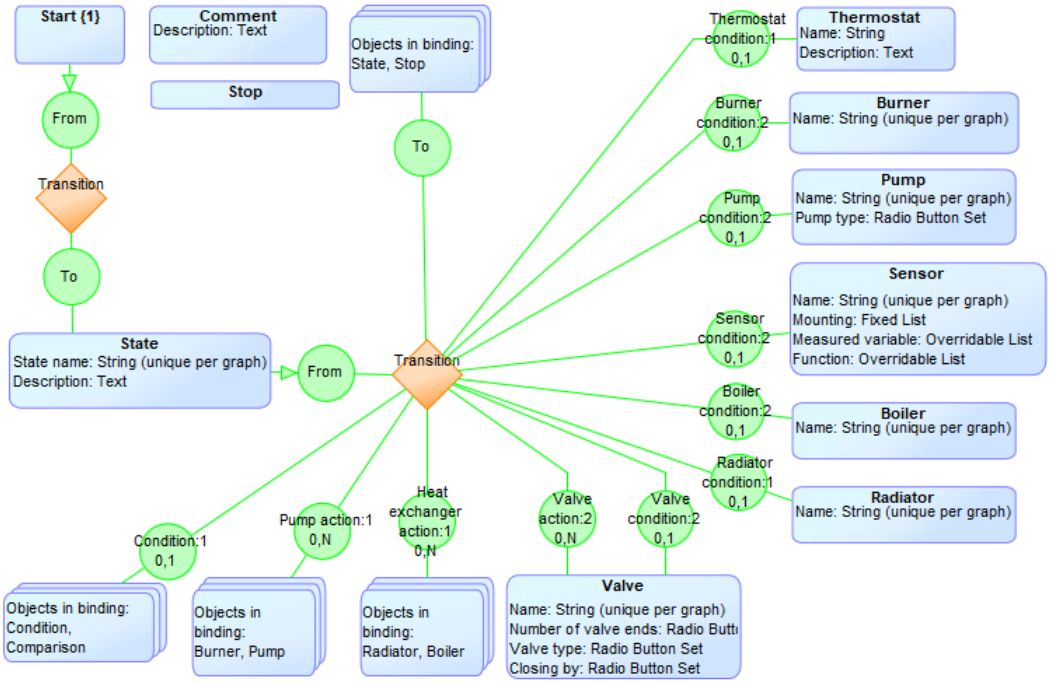


Figure 2-1. Metamodel of heating application language (partial)

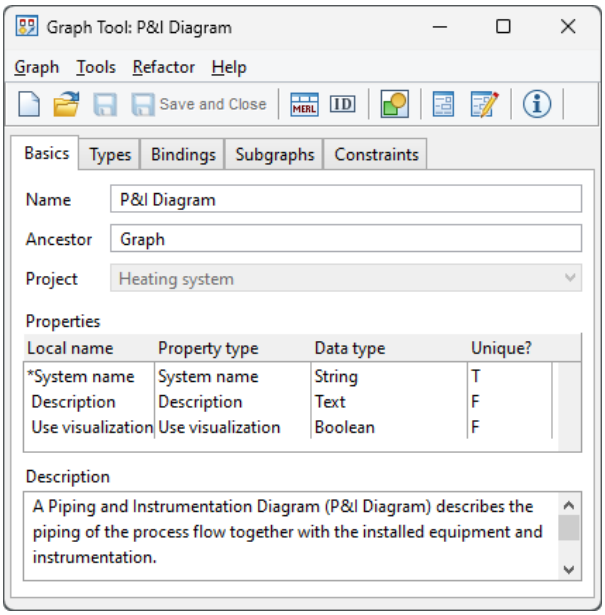
After the above metamodel is drawn in the Diagram Editor of MetaEdit+, it can be instantiated and tested immediately with the modeling editors providing full editor functionality (copy/paste, undo, replace, trace, print etc.). Once the language definition is complete it can be given to the developers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

2.1.2 Form-based metamodeling tools

The form-based metamodeling tools can define the abstract syntax of the language, along with its rules and constraints. notation and generators for model checking, code generation, documentation generation etc. The form-based tools are integrated with the other tools in MetaEdit+, allowing the metamodeler 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.

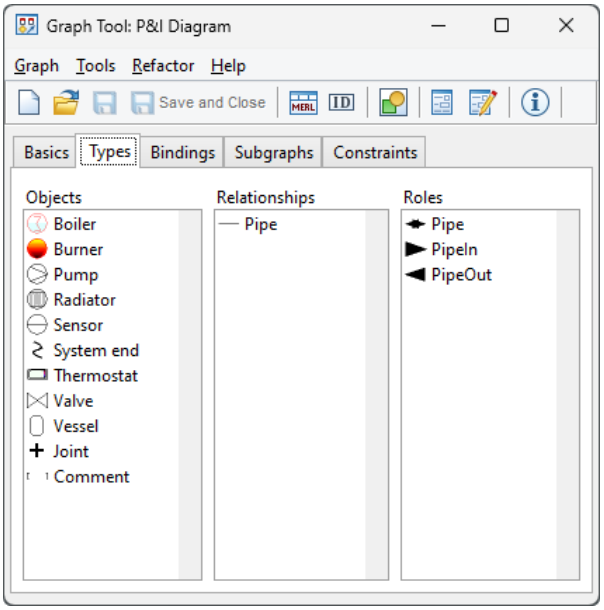
Most importantly, the definition of language elements is automatically applied in the various modeling editors (Diagram, Matrix, Table), browsers (Graph, Object, Type) and generators. This supports agile and incremental language definition: updates to the modeling language can be tested immediately and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

Language concepts (abstract syntax)

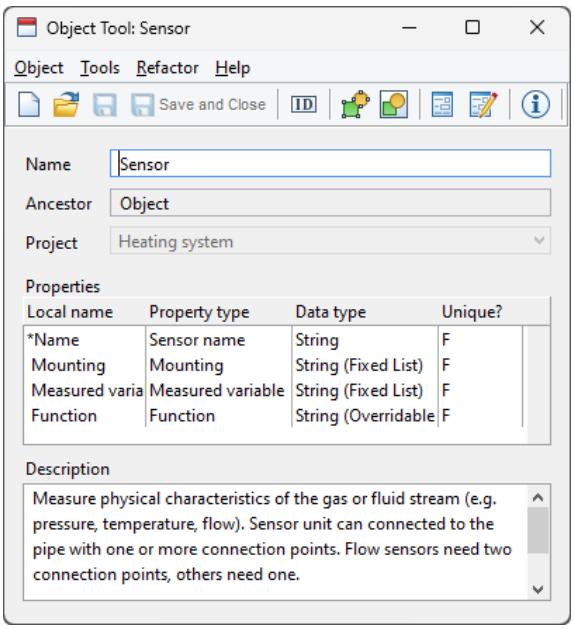


The Graph Tool defines the individual languages. Here a definition for a 'P&I Diagram' is given. A diagram itself has properties, and each property has a more detailed definition. For example, each P&I Diagram has a 'System name', which is its identifying property (marked with '*'). It is of data type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

The Graph Tool also 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…**).



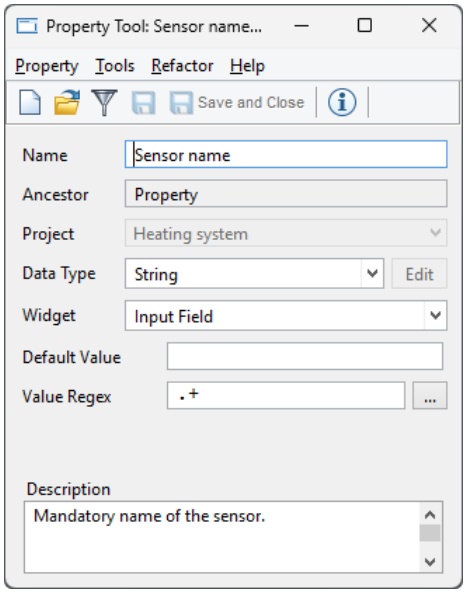
The Types tab in the Graph Tool shows the individual language concepts: object types, relationship types and role types. These different kinds of concepts can be added or removed here from the language.



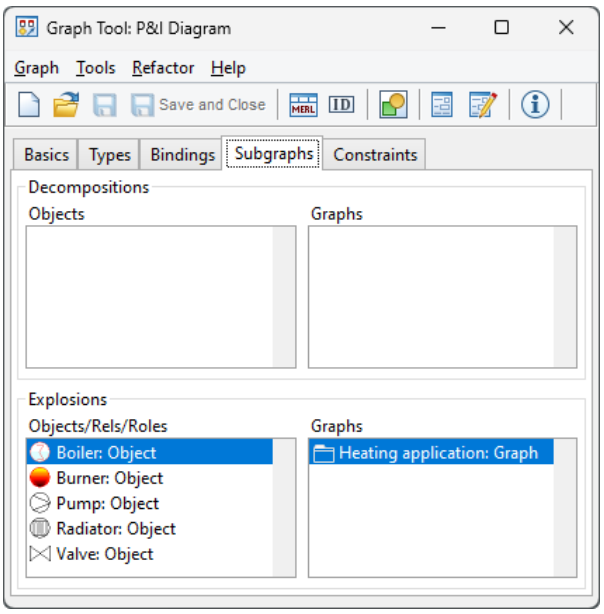
For each type, a form-based metamodeling tool shows its definition. Here an Object Tool shows the definition of 'Sensor', an object type. The metamodeler has specified the type's Name, Supertype, Project, four different property types, and a Description.

The property dialog for entering values for instances of the type is automatically created based on the properties defined; it can also be manually laid out if desired.

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



In the Properties list for Sensor (above), we see that a Sensor has a property, whose Local name is 'Name', and whose Property type name is 'Sensor name'. The definition of the 'Sensor name' property type is shown here in a Property Tool. It is of String data type, uses a normal 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 ('.+').



The Subgraph tab in the Graph Tool sets which types of element in this type of graph can have a subgraph, and of what type. The subgraph may be of a different type from the parent graph, as in the case of heating systems, for example: a Boiler in a P&I Diagram can have its behavior described in an explosion subgraph of type 'Heating application'.

Constraints are specified with the form-based tools as a part of the metamodel: the basic legal structure of relationships is specified on the Bindings tab, and extra constraints can be specified on the Constraints tab. More complicated rules or those that should be checked only on demand can be specified with generators. Below some 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, each 'System end' may have only one 'Pipe' relationship connected to it.

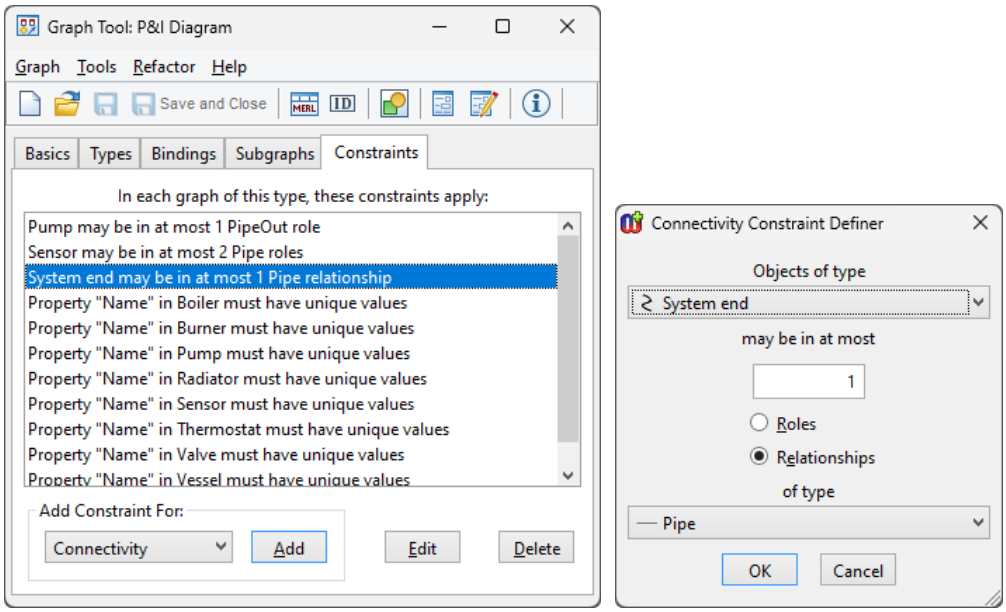


Figure 2-2. Defining constraints of the language

2.1.3 Notation

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

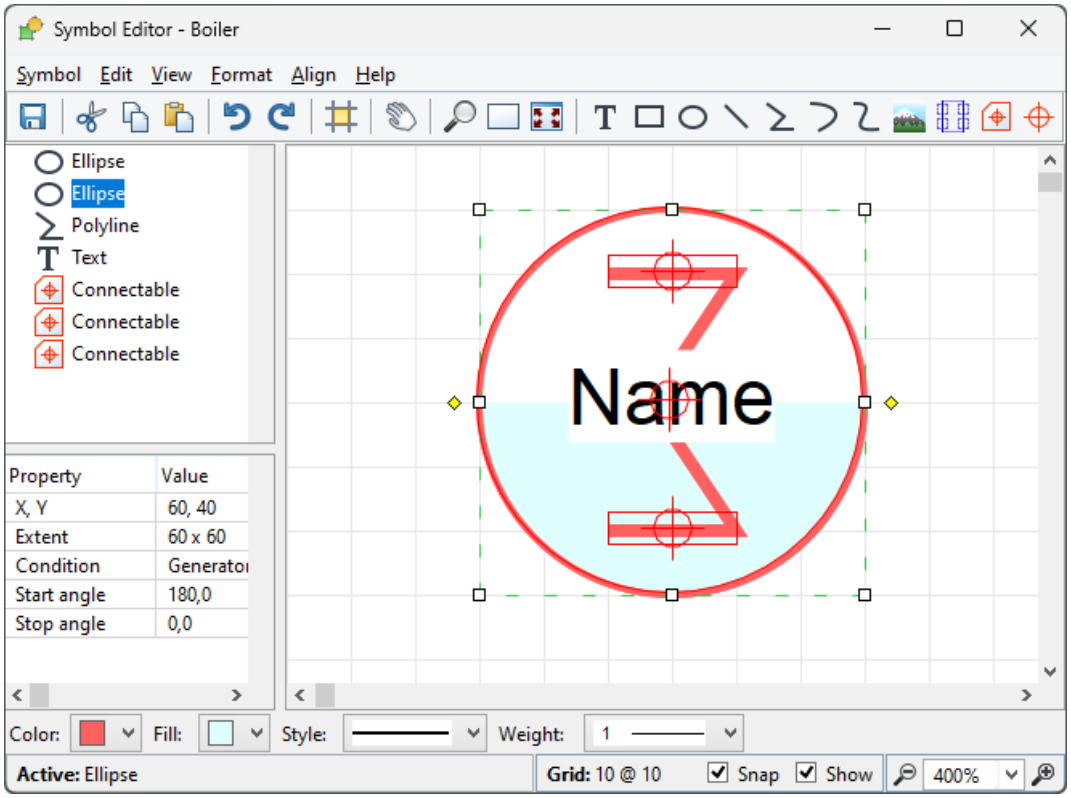


Figure 2-3. Symbol definition for the Boiler

A symbol consists of multiple symbol elements, and each symbol element can be drawn here as a vector graphic element like a rectangle, ellipse or freeform polygon, or as an imported bitmap. The symbol of 'Boiler' consists of two ellipses, a polyline, and a text label showing the value of the Name property of the Boiler. The symbol definition also shows three red connectables, each with a crosshair for relationships' role lines to aim at, and a thin red outline at which the incoming line stops. 'Boiler' has a connectable for 'Pipe' connections for cold water coming in at the bottom, and another for hot water going out at the top. The third connectable in the middle is the default, used to draw connections for instruments controlling the boiler (or for a separate flow, e.g. hot water to taps rather than central heating radiators).

Existing symbols or partial symbols can be imported from MetaEdit+'s Symbol Library, or from SVG vector graphics files. Individual symbol elements can be made conditional depending on the model data, and textual symbol elements can obtain their contents from fixed text, property values, or the results of generators. A template element can be used to make parts of the symbol repeat or to include subsymbols based on other elements in the model.

The icon used for a given language concept in lists, buttons and menus is automatically produced by scaling down the concept's symbol. It can also be overridden by the metamodeler by drawing a bespoke icon for that concept in the Symbol Editor.

2.1.4 Generators

The MetaEdit+ Generator System is used to define generators for a variety of needs. In the case of heating systems, generators are used to produce code, model checks, a hardware installation guide in HTML, Word documentation of the system, and to send the generated code directly into the TwinCat environment.

MERL, a domain-specific language for generator development, is used to define generators. The figure below shows a Generator Editor for P&I Diagrams, with the MERL definition for function block generation. The generator script is shown in the bottom half of the window, a hierarchy of generators at the top left, MERL templates in the top center, and the concepts of the language (the metamodel) at the top right.

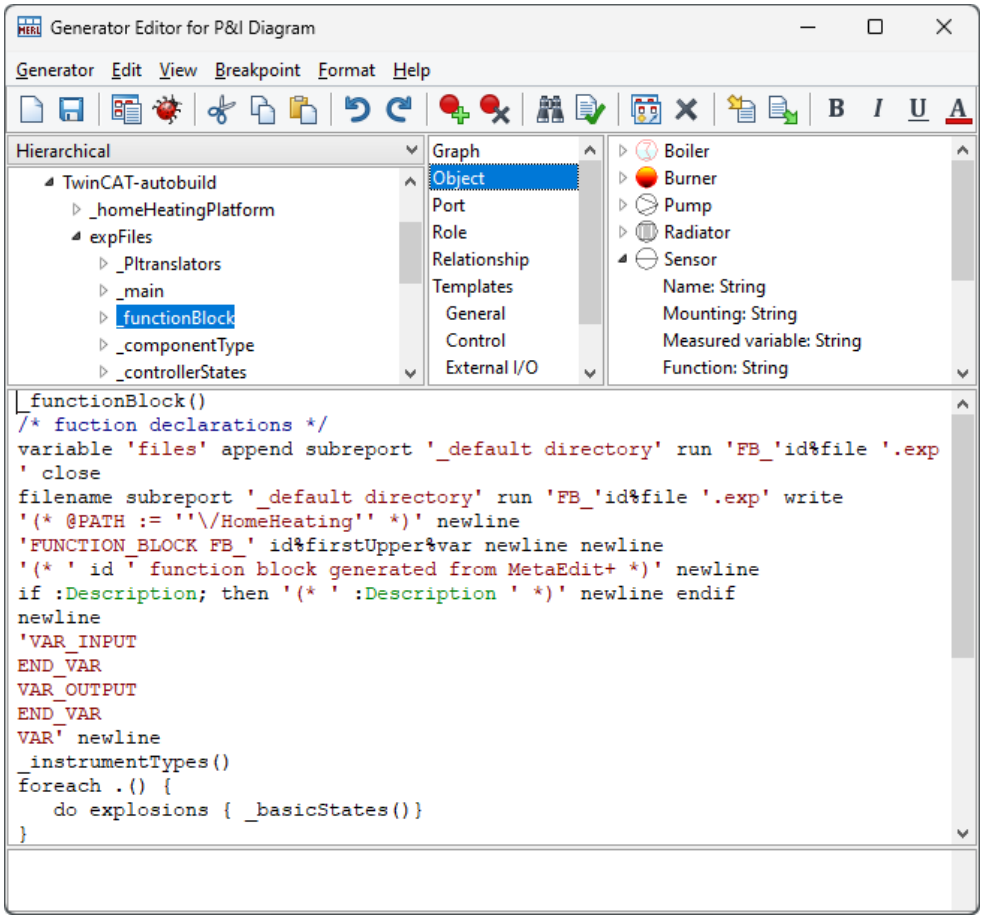


Figure 2-4. Generator definition in MERL

The Generator Editor supports syntax highlighting, static code analysis, error detection, navigation among subgenerators, and content assistance for showing the metamodel concepts and inserting their names in the generator. MetaEdit+ also provides a full MERL source-level debugger with breakpoints, interrupts, conditional breakpoints, live editing of variables etc. In addition to the Generator System, MetaEdit+ offers an API based on web services/.Net/SOAP for accessing the model elements using other systems.

As well as being available for running via menu operations, generators can also be added to the action toolbar of model editors, and a custom icon can be defined for them in the Symbol Editor. Generators can also be used for individual language concepts, to control how they are displayed in lists and texts, e.g. by combining a First Name property, a space, and a Last Name property.

2.2 DEFINING A P&I NETWORK OF THE HEATING SYSTEM

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

The 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 do not have any cover are shown with thin solid lines. This visualization was defined with the Symbol Editor.

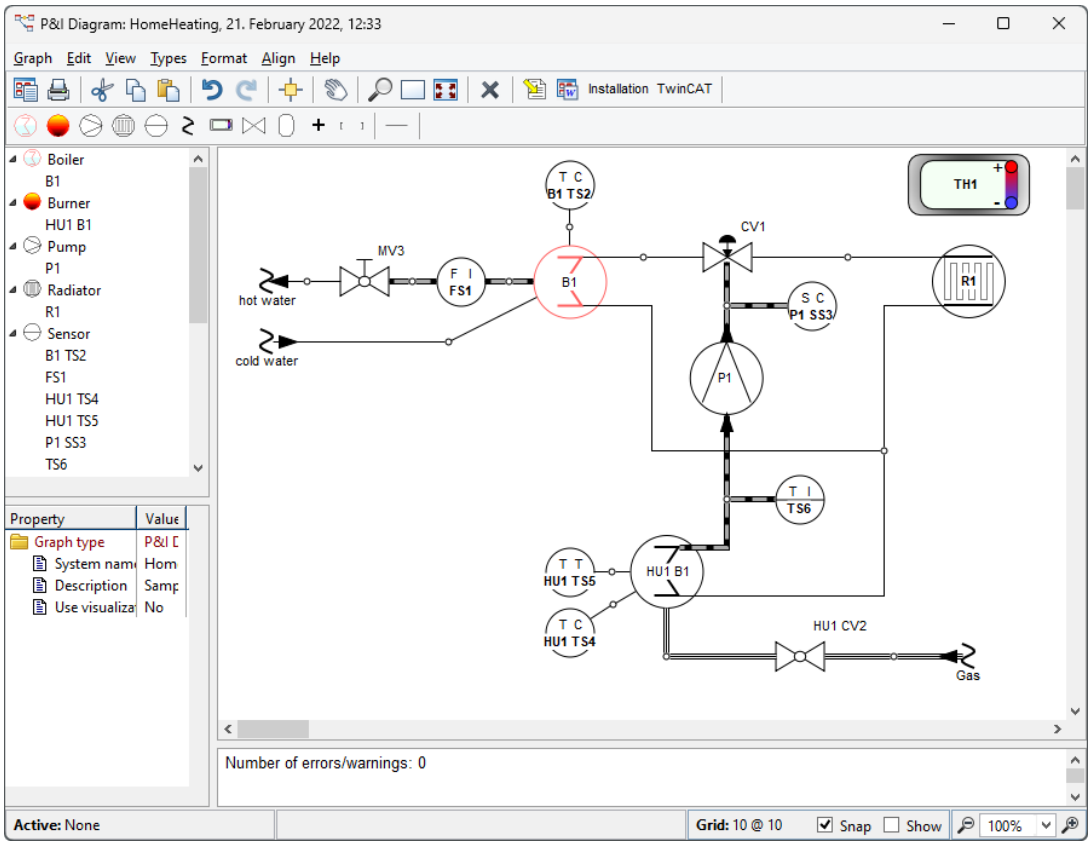


Figure 2-5. Instruments and their pipe connections

The Diagram Editor shows the description of the LWC2012 heating system using the DSL. The second toolbar row shows the main concepts of the DSL. Models elements are created by clicking the domain concept on the toolbar and then clicking in the diagram to add it. During model creation and editing the Diagram Editor checks that the DSL definition is followed: both by checking the definition of the metamodel and by running the checking scripts written in MERL.

2.3 IMPROVED VISUALIZATION OF THE P&I NETWORK

The notation shown above follows the well-known and widely-used notation of Piping and Instrumentation, but we can also enhance it with improved coloring and error annotation. The figure below shows the same heating application using these visualization enhancements. These visualization options can be set on or off from the graph's property 'Use visualization'.

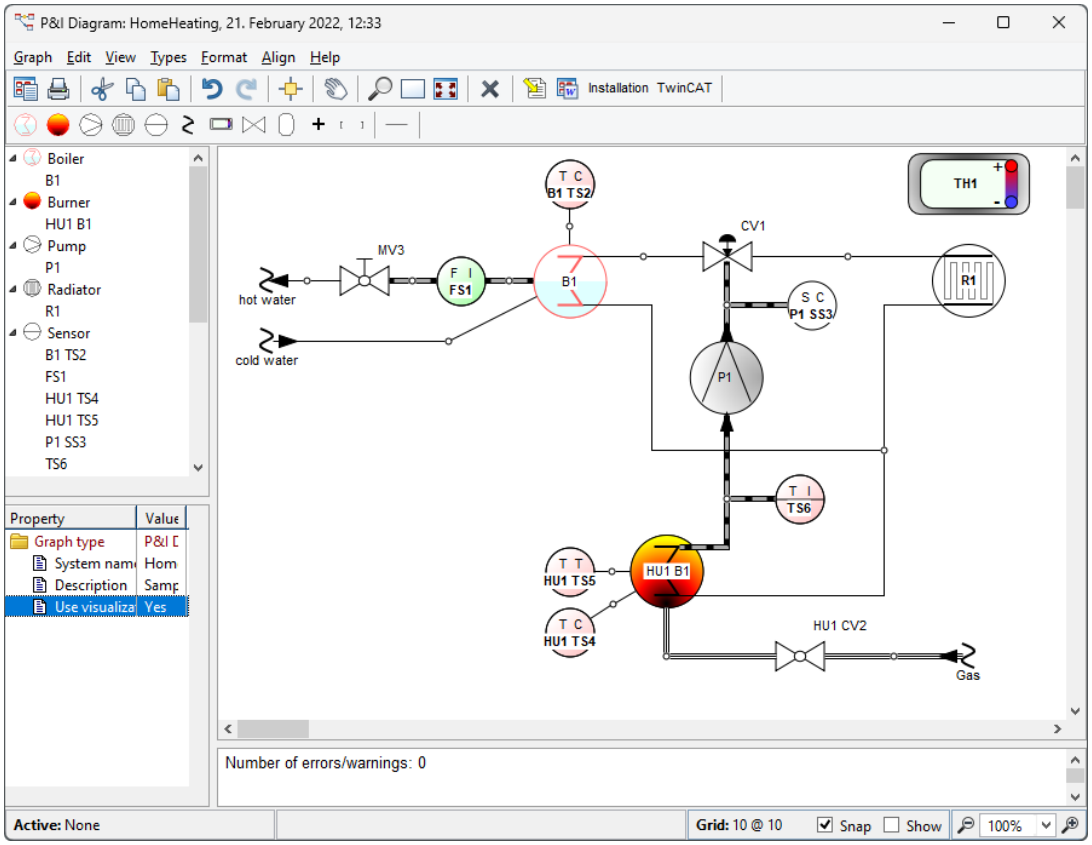


Figure 2-6. Visualizing instruments in the P&I Diagram

The editor also shows possible errors and incompleteness information as defined by the language engineer. Here two errors/warnings are reported: the FS1 Sensor needs a second pipe connection, and the MV3 Valve also needs one or more extra pipe connections. Double-clicking the elements in the error report selects the corresponding element in the diagram, allowing you to update it easily.

While the P&I network is shown here 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. The Diagram Editor below shows the behavior of the ‘Pump’ *P1*. The language is based on a state machine, but is domain-specific in that it only offers access to conditions supported by each instrument (like ‘Boiler’s’ flame detection), and to the actions available for each instrument (like turning a ‘Pump’ on or off).

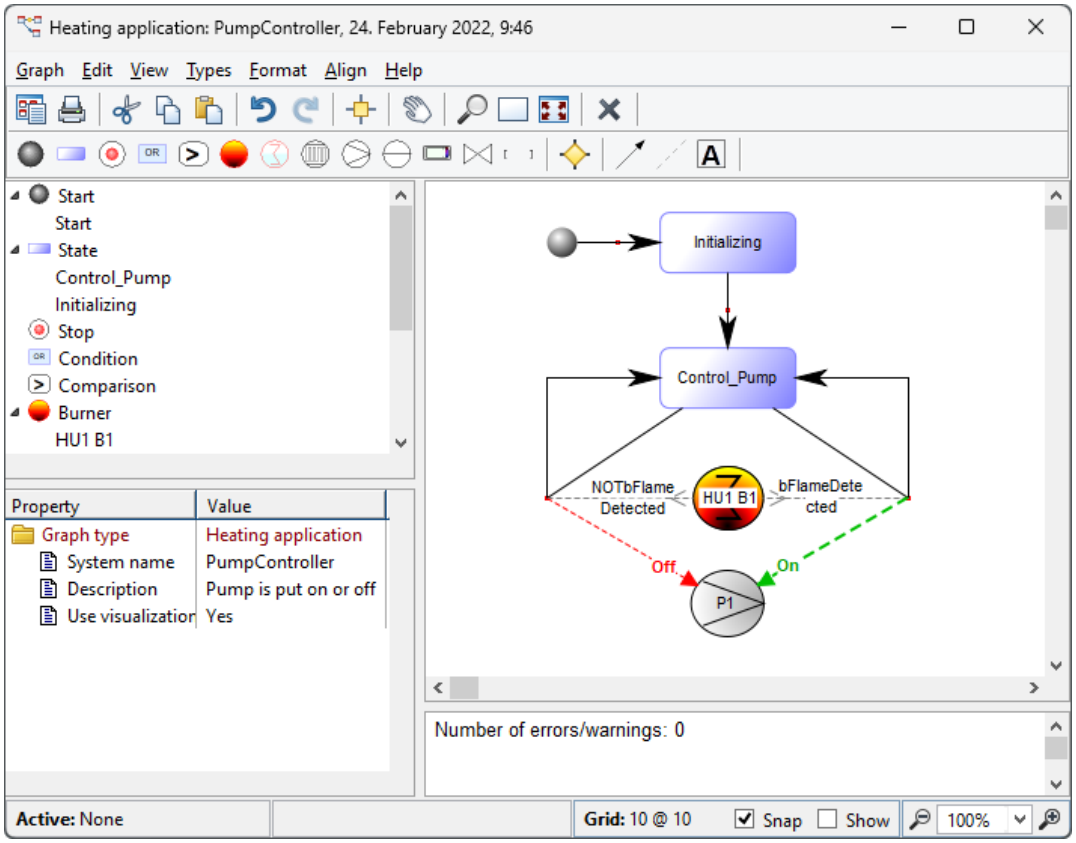


Figure 2-7. Behavior of pump controller

The behavior model refers directly to the instruments defined in the P&I Diagram. In other words, a given instrument is the same in both kinds of diagrams – no need to update a name in multiple places or keep other values in sync. The Diagram Editor (as well as other editors and browsers) allows you to trace between these two kinds of diagrams as well as see how instruments are (re)used. By selecting **Info…** from the element’s pop-up menu you can see the places it is used. The languages also support consistency checking across the different diagrams. For example, if a behavior model uses instruments that are not defined in the 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 its own constructs, available from the toolbar like all other language constructs. Below, a diagram for Heat Controller shows a constraint based on ‘Radiator’ *R1* and ‘Boiler’ *B1* warming. This example also shows entry and exit actions. While they are not necessary, as the same functionality could be specified via state transitions too, they were added to the heating application language to mirror the reference implementation more closely.

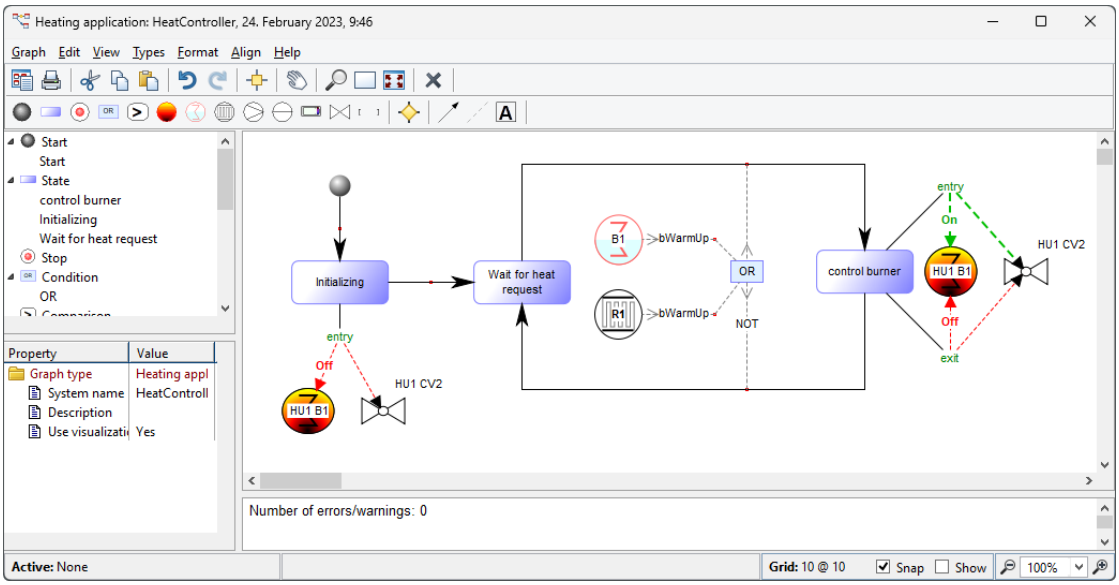


Figure 2-8. Behavior of heating controller

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 the language user always follows the constraints — even without knowing about them. Therefore the interlock/constraint definition can be supported as in Section 2.1, defining language elements.

If there is a need to model the interlocks/constraints explicitly and per system, then one solution would be to add a new DSL with concepts for interlock. That DSL would still be integrated with the other DSLs. An example of an invariant is that if the burner is on then the pump must be running. Below such a rule is defined along with an emergency shutdown action: if the pump is not on and the burner is running then the burner is turned off. Other constraints could be added to the language too, but here the already available DSL constructs can be used to describe invariants/constraints.

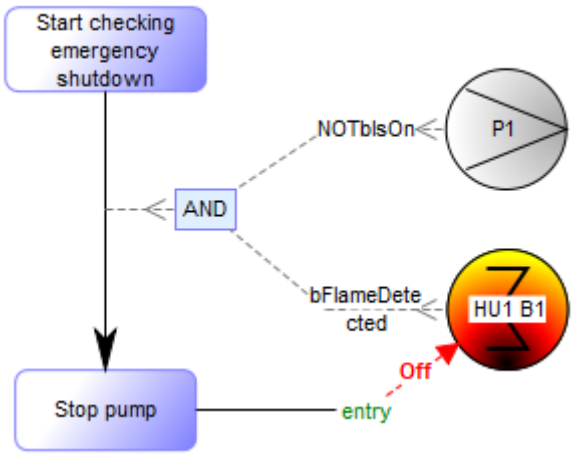


Figure 2-9. Modeling invariant/interlocks with the DSL

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 the whole code generation process available from the toolbar of the P&I Diagram. A sample of the generated data type code for a 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. The figure below shows the generated installation guide, which lists all the instruments needed and also calculates how much pipe is needed.

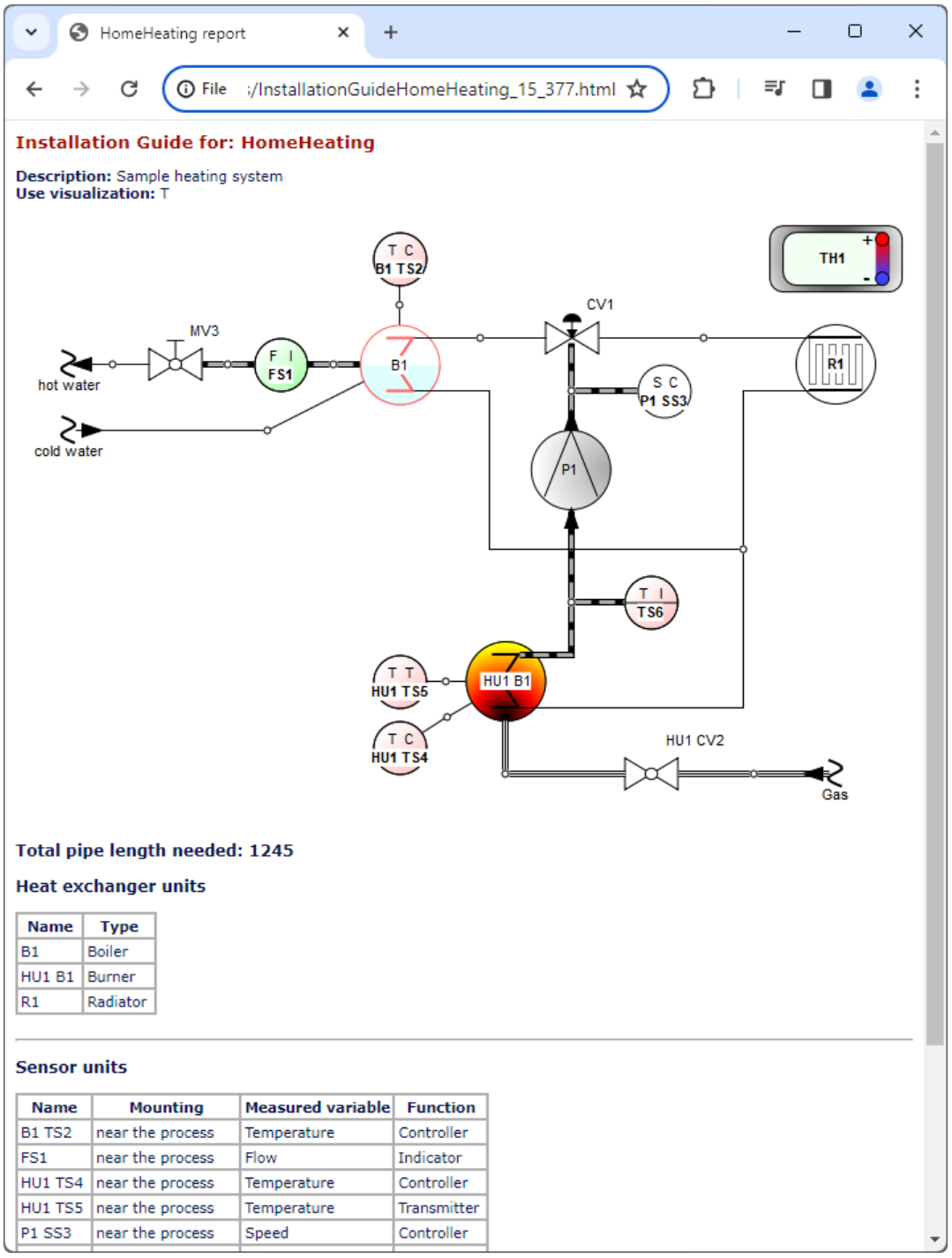


Figure 2-10. Generated installation documentation

2.7 GENERATING CONTROL CODE FOR A TARGET

All the code needed for the heating application, both for its structural definition and for the function blocks specifying behavior, is 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 run the generators choose **Graph | Generate…** and select 'TwinCAT-autobuild', or press the 'TwinCat' button in the toolbar of the Diagram Editor for a P&I Diagram. The code generated with TwinCAT-autobuild uses the platform of the home heating 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 normal practice the platform would more likely be a separate project file in TwinCAT. The autobuild generation starts TwinCAT, telling it to import the generated code and compile it for execution. When creating the project TwinCAT asks to choose the target system. Select PC. If you just want to see the generated code, run the 'expFiles' generator.

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

The 'TwinCAT-autobuild' and 'expFiles' generators also produce 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 THE INTERFACE FOR HIGHER LEVEL SOFTWARE

Interface code is generated similarly to other generator output of MetaEdit+: MERL generators access the same heating application models. The example interface code generated creates high level operations 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 an 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, and you can try this out with the code generated from MetaEdit+ too. The figure below shows a snapshot of the simulation for the generated heating application. Here the visualization elements are those available from TwinCAT.

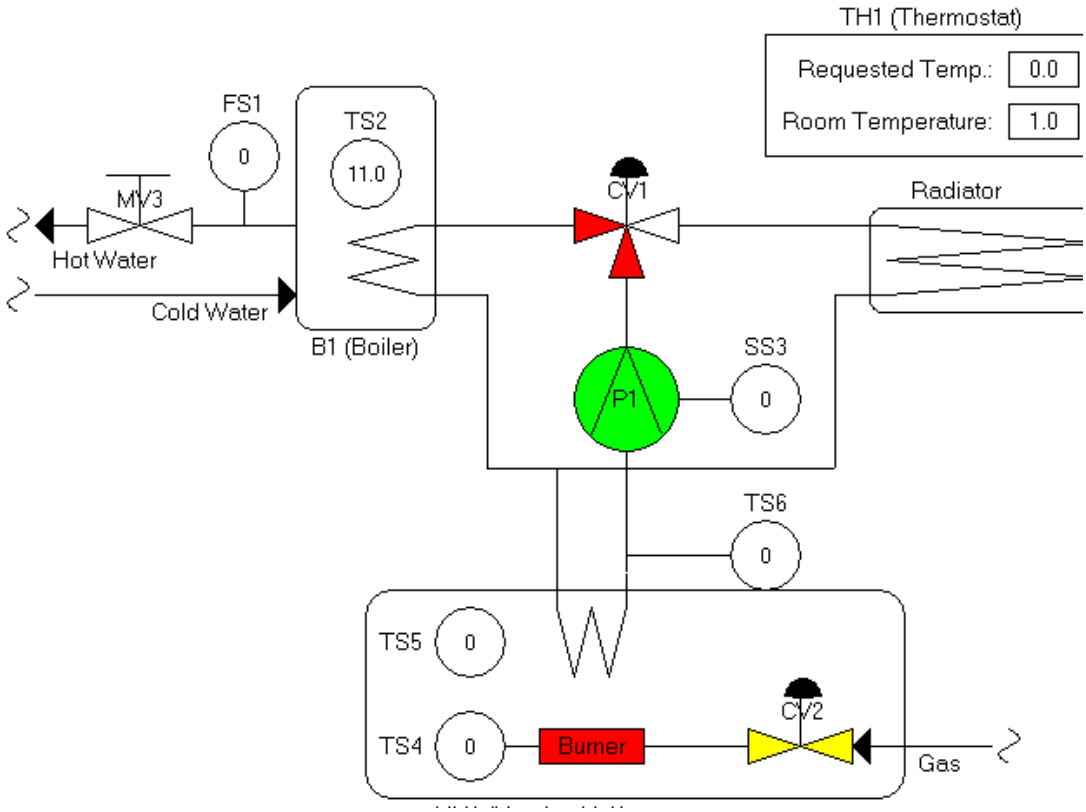


Figure 2-11. Visualizing the heating system in TwinCat

An alternative approach would be to use MetaEdit+ directly to visualize the execution. In other words, MetaEdit+ animates the 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 the Diagram Editor of MetaEdit+. MetaEdit+'s API is based on the SOAP / Web Services / .NET standard for application integration, available for almost any programming language and scriptable 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. The MetaEdit+ API is not restricted to animation only: it can be used to read, modify, create and delete the elements in models, and also the representations of those model elements.

3 Conclusions

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

On the DSM definition side we focused on a few areas of language design: integrating two languages, generating the code so that it integrated with an existing PLC development tool, and producing various other artifacts like an 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 kinds of code than the current state-based PLC code. The choice is yours because with DSM you control both the language and the generators.