

AtomWeaver



Piping & Instrumentation



Language Workbench Challenge 2012

Table of Contents

PART I : ABSE Introduction.....	3
About ABSE.....	3
Decomposition, Composition, Association, Context: Getting closer to our brains.....	3
ABSE Models.....	4
How it works.....	5
Lua, ABSE's transformation and execution language.....	6
Cooperative meta-programming.....	6
The generation process.....	7
Creating systems using building blocks.....	7
Reuse means productivity.....	8
Support for ALM (Application Life-cycle Management).....	8
ABSE is universal.....	9
Any language.....	9
Any development methodology.....	9
Any domain.....	10
Refactoring.....	10
Generated code and custom code co-existence.....	11
Large model support.....	11
Traceability.....	12
PART II : AtomWeaver Introduction.....	14
About AtomWeaver.....	14
AtomWeaver architecture.....	14
PART III : LWC 2012 Assignment.....	16
The State Machine Atom Library.....	16
The IEC 61131-3 Atom Library.....	17
The Home Heating Atom Library.....	18
Controllers.....	18
Connections.....	19
Components.....	19
Solution Composition.....	20
The Home Heating System metamodel.....	21
Modeling the Home Heating System.....	23
Starting out.....	23
Separating the installation into physical spaces.....	25
Creating the water/gas supply points.....	25
Creating the rest of the Machine Room's components.....	26
Creating the Bedroom's components.....	27
Connecting two components.....	27
Completing the system's circuits.....	30
Checking connections.....	30
Generating Code.....	31
Conclusions.....	31
Reuse in action.....	31

PART I : ABSE Introduction

If you are having your first contact with ABSE, the following pages give a quick introduction to the framework's concepts and capabilities:

About ABSE

ABSE stands for “Atom-Based Software Engineering”. It's a software development automation framework with the aim of supporting the complete application life-cycle through models and code generation.

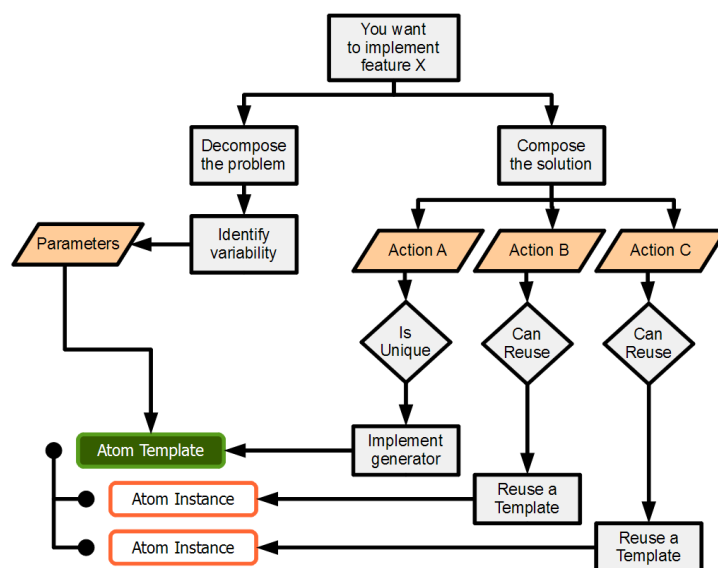
ABSE implements a software development methodology that reflects what software development is about: problem decomposition and solution composition. It shifts the focus away from the computer's code and into ideas and concepts.

You can use ABSE to model and generate complete applications, but you can also build just a portion of an application. You can also generate text or data files in any format, web sites, or any combination of these.

Decomposition, Composition, Association, Context: Getting closer to our brains

Everyday we solve problems, and software development is all about solving problems. You usually decompose requirements into smaller requirements, and therefore, problems into smaller problems. Then, you find a small solution for each of those small problems. This is how we do it, as far as theory goes.

In addition, we can build larger solutions by combining smaller solutions together in an orchestrated way. If we do this on a permanent basis in real life and especially in software development. When implementing a concept/feature with ABSE, you'll roughly follow this workflow:



This is just one example and the number of composition actions, and their nature, will vary.

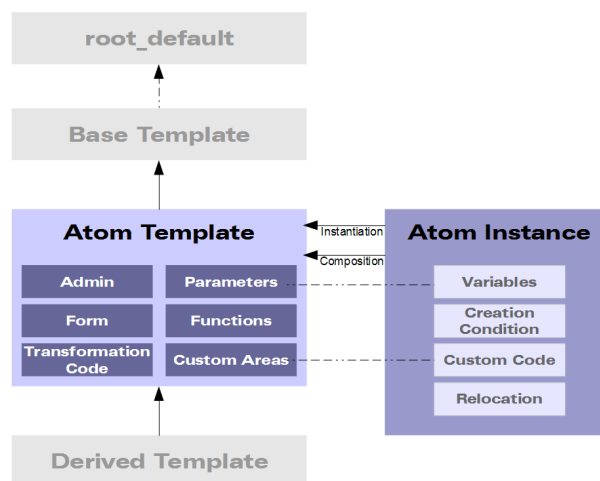
So when we are developing, we are constantly splitting problems into smaller problems, or composing solutions into larger solutions. Your permanent questions will always revolve around something like “How is this done?” (problem decomposition), or “What is this made up of?” (solution composition).

When you model through ABSE, you must answer these questions when you are building Atom Templates, and then again when you build your project model.

ABSE Models

ABSE models are always trees. A tree conveys association and context, very close to how our brains work. That is why any non-trivial application relies on a tree in one way or another. ABSE can also link any two Atoms together, allowing you to emulate any kind of graph.

ABSE's meta-metamodel, the Atom, defines how its three manifestations (Atom Template, Atom Organizer, Atom Instance) work together to build your libraries and projects. Here's the most important portion of the Atom meta-metamodel:



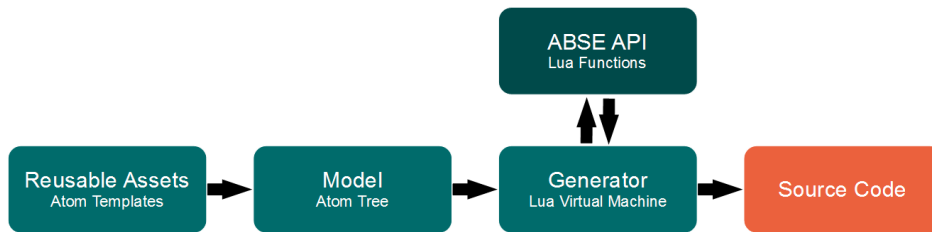
ABSE relies on metadata and transformation code that are grouped together to form what is called an Atom Template. These will be your reusable assets. A library of ready-made Atom Templates will help you reuse all your knowledge in a quick and thoughtful manner.

To build your project through reuse, you compose it by instantiating your Atom Templates as if they were building blocks. This is called *software composition*. These new Atoms, called *Atom Instances*, are organized onto a tree to form a model of the artifact(s) you want to build.

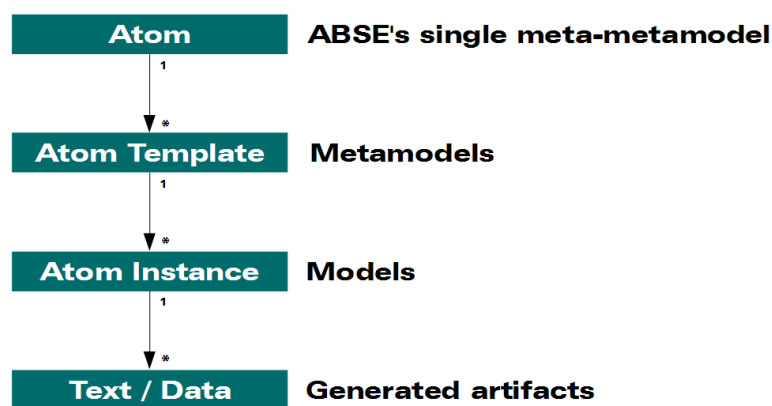
ABSE is in some way comparable to OOP (Object-Oriented Programming): Atom Templates are comparable to classes, and Atom Instances are comparable to the instantiation of those classes. Inheritance and composition are also similar concepts. This means that most of ABSE's concepts are immediately familiar to developers.

How it works

The ABSE framework is composed of an API, an execution language (Lua), and a code generator:



In a model-driven development context, ABSE provides the following abstraction levels:



Because ABSE relies on reuse, anything new you want to create on an ABSE project you will have to create first on an Atom Library.

Lua, ABSE's transformation and execution language



Because ABSE relies on an API, it does not need a specially tailored DSL. Instead, it uses Lua, a general-purpose language, to specify Atom Template's properties and transformation code.

Lua is an open-source, lightweight, reflective, imperative and functional programming language, designed as a scripting language, and is compact enough to fit on a variety of host platforms. By including only a minimum set of data types, Lua attempts to strike a balance between power and size.

Lua is a small language, and the subset of Lua features that you need for ABSE is even smaller.

Cooperative meta-programming

An Atom Template is made of several *Sections*. Each Section is actually a Lua mini-program. When modeling or generating, the host modeler (AtomWeaver) calls these mini-programs for each instantiation (Atom Instance) of that Template.

Each Atom Instance has the responsibility of building a small part of your final system, while cooperating with other Atoms.

This approach brings ABSE into the meta-programming category.

Effectively, the host modeler (AtomWeaver) will weave a program from all project Atoms and run it to obtain the desired generated artifacts. Therefore, we will have a program generating another program, qualifying it as a meta-program.

The generation process

The code generation process is segmented into four phases. The first phase (*Create*) is run by the host modeler at Atom creation time and is comparable to an object constructor.



The other three phases (*Pre*, *Exec* and *Post*) are run by the Code Generator. The *Pre* phase lets you prepare the code generation process. The *Exec* phase performs the actual code generation, and the *Post* section lets you generate code on top of already generated code.

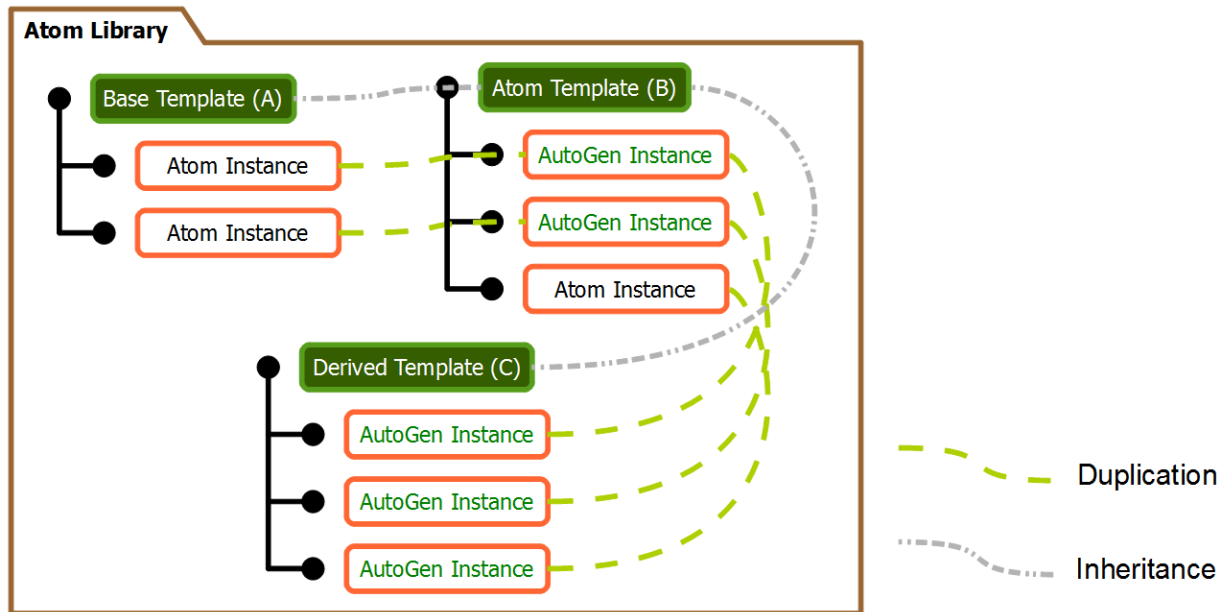
Creating systems using building blocks

Most of us developers have dreamed of building software applications by just snapping small parts together, just like the Lego bricks we all have played with in our childhood.

However, albeit constantly sought for, this goal has never been quite reached. The extremely broad application of software engineering makes it quite difficult to create infinite possibilities from components that have a finite, small number of “connections” or options.

But with the emergence of domain-specific modeling, we can narrow those “infinite possibilities” into a manageable “large quantity” of possibilities, making it possible now to model a system that is complex, but within a known domain, using those aforementioned building blocks.

ABSE innovates on this front: Atom Templates (ABSE’s metamodels) can be made of other Atom Templates, creating larger components. And through Atom construction constraints, you can define to which other “blocks” can a “block” snap to. These constraints implement model construction guidance.



The above diagram shows the inheritance chain of three different Atom Templates. An Atom Template always inherits the base Template's composed Instances: Template A is composed of two Instances. Template B inherits and extends Template A: It inherits and duplicates Template A's Instances, and adds one of its own. Finally, Template C inherits from Template B, again duplicating its Instances.

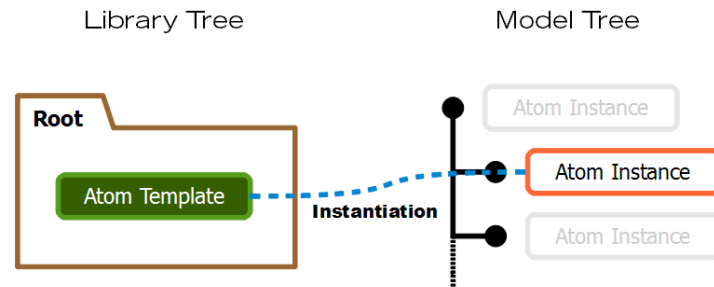
Correct models

ABSE can be used as a pure "snap together" modeling system. Because construction constraints are checked at creation time, it is guaranteed that the architecture of the developed system is always correct. This will help less experienced developers work like the expert that defined the system's architecture. Descriptions and other Atom Parameters that the expert modelers might wish to add will guide the developer in the model discovery process.

Reuse means productivity

ABSE is heavily focused on reuse. Anything you want to build on an ABSE Project Model you will have to create first on an Atom Library. This looks like an additional cost over traditional programming at first, but you'll notice the productivity improvement as you quickly and efficiently reuse all your accumulated code and knowledge.

An Atom Template is in fact a pure reusable piece of knowledge that you can combine into larger components and systems to obtain your desired runnable code and data. When you reuse an Atom Template, it becomes an Atom Instance in an ABSE Tree:



Using metamodels (Atom Templates in the case of ABSE) brings several added benefits that influence productivity, like:

- Quickly reusing existing knowledge (code or other artifacts)
- Reusing tried and tested code instead of doing it again (and risk introducing new bugs)
- Enforcing best practices and architectures
- Model constraints guide newbies and avoid architecture fragility and spent time refactoring

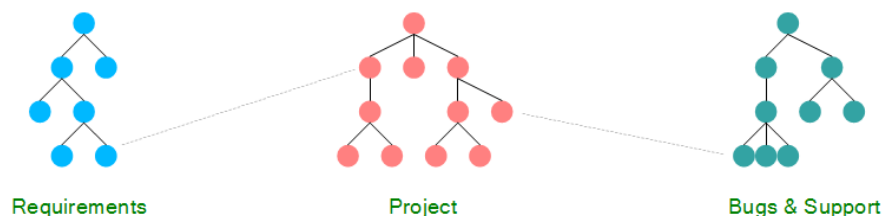
Effective reuse is the holy grail of software development ROI. Each time you successfully reuse your knowledge, you are doing more with less, and that's the way to go if we want to tackle the ever-increasing need for software with a finite work force.

Support for ALM (Application Life-cycle Management)

Application Lifecycle Management (ALM) is normally performed by specialized integrator software that orchestrates the multiple phases in the life of a software product. These products are needed because the development approaches in use do not cover the whole development spectrum.

An ABSE project can have an unlimited number of trees. And one Atom Instance in one tree can reference another Atom in another tree. Specific trees have specific objectives: You can create an ABSE tree for requirements capture, another for your software application, another for issue tracking, another for end-user documentation, and so on.

For instance, an Atom that implements a certain program feature can be linked to its corresponding requirement Atom; The same feature Atom may generate a piece of end-user documentation by automatically adding an Atom on the documentation tree.



ABSE is universal

ABSE is not targeted at any specific programming language or hardware platform. In fact, you can target multiple platforms and generate code in any number of programming languages at the same time, even on a single Atom. When technologies come and go so quickly, you can safely invest on

your ABSE metamodels.

Any language

You can command any Atom Template to generate free-form textual output. Therefore, you can generate any kind of text file, in any format you need. You can generate into multiple files, of different types, from a single Atom.

For example, you can model a RIA architecture where you can generate, from any single Atom, web pages (HTML), design (CSS), definitions (XML) and code (JavaScript, ASP.NET, others).

While this might seem a source of confusion, that might occur by mixing languages and platforms, ABSE has separation mechanisms that keep your projects from getting messed-up: Atom Libraries, Atom composition and Atom Template inheritance.

Any development methodology

Because ABSE is closer to our brain logic rather than a specific development methodology, it can support all software engineering methodologies.

If you want to implement Component-Based Development (CBD), you can directly use Atom Templates since they are comparable to components.

If you consider an Atom Template as a feature, you can then implement Feature-Driven Development (FDD). An Atom Template can be composed of other discrete components, snippets, models or sub-features.

You can easily weave aspects to your application classes or features, which makes Aspect-Oriented Programming (AOP) an intrinsic characteristic of the whole ABSE approach.

If you want to work at a higher abstraction level, in the problem domain, you can implement Domain-Specific Modeling (DSM). For instance, you can model your solution domain Atom Libraries first, and then reuse Atoms on those libraries in higher-level, problem domain libraries.

If do not want to implement any specific software engineering approach, you can still develop you application using the traditional line-oriented, file-based development by just implementing very simple models of your files and object.

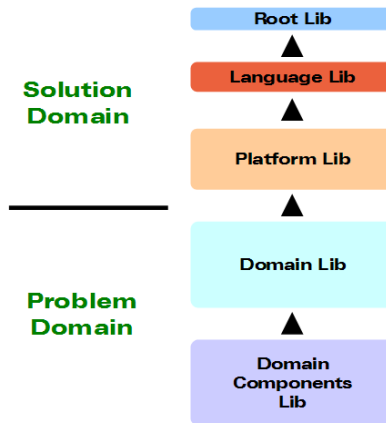
And of course, you can mix any and all of the above approaches with your own...

Any domain

Any domain can be modeled with ABSE. Today we see that domains cross over all the time. For instance:

- Automotive has built-in entertainment (a problem domain cross-over)
- Entertainment devices have web access (a problem-solution domain cross-over)
- Web apps use databases (a solution domain cross-over)

So, in a world where domains mix and cooperate most of the time, an approach that supports any domain stands a better chance of evolution and long-term stability. Atom Libraries can reuse Libraries from different domains or at lower abstraction levels:

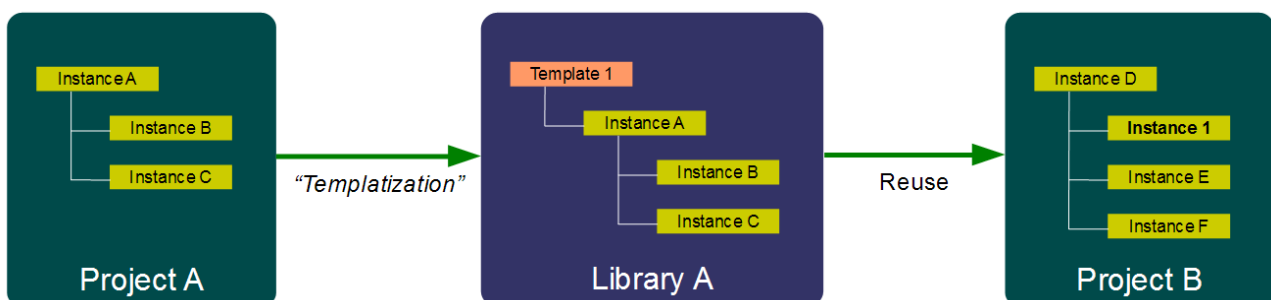


Refactoring

Refactoring in a traditional programming approach is the process of changing a program's code without changing its functional behavior. Like in traditional programming, you can refactor your ABSE models. And the same benefits apply.

One simple example is when you have a new piece of code. Because it is new, you just create a custom-code Atom to include it in the project. However, as repeated usage of that code segment is detected, it can be turned into a specific domain Atom.

Another example is when a pattern of Atoms is consistently used many times over: This pattern will be regrouped as a new Atom Template (a new meta-model) which can then have its own parameters and construction constraints, probably turning those low-level, solution-domain Atoms into one high-level, domain-specific Atom.



The pattern on Project A is turned into an Atom Template, making it possible to be reused on the same or other projects.

Generated code and custom code co-existence

ABSE models support the notion of a “custom code block”. Code blocks are first-class citizens: You can supply a piece of custom code as a parameter to an Atom Instance. It's then up to the metamodel (Atom Template) to decide what to do with that piece of custom code. Usually it is simply placed in a specific section on a generated file, but the metamodel can do additional processing on that block.

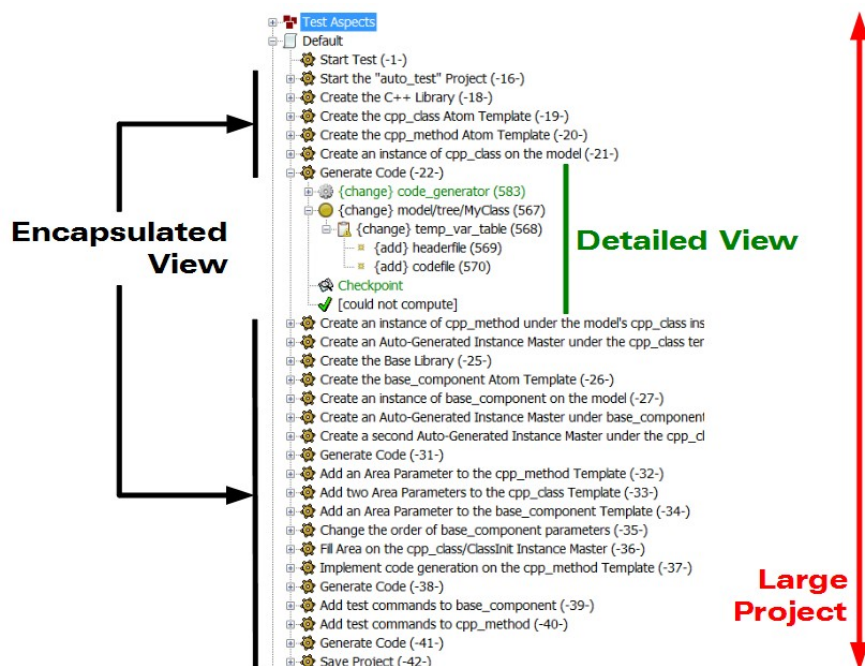
Management of custom code blocks by the model itself avoids these common problems usually found when mixing generated and custom code:

- Mistakenly edited generated code is lost on regeneration
- Regeneration deletes your custom code
- Problems distinguishing generated from custom code
- Last-minute, problematic hacks on generated code

Large model support

An ABSE model is represented by a tree and not by a graph. Graphs are prettier to the eye but have severe layout and space requirements when large feature densities are reached: If want to have a global view of the project, you have to give up on the details. On the other hand, if you want to see some details, you have to give up the global view.

An ABSE tree can have millions of Atoms and still be manageable. Why? Because a tree is a fractal object: Successive details can be uncovered by successively opening deeper branches. Conversely, you can progressively encapsulate smaller details by closing tree branches. You can be looking at different levels of the tree but still seeing just the right amount of detail for the task in hand.

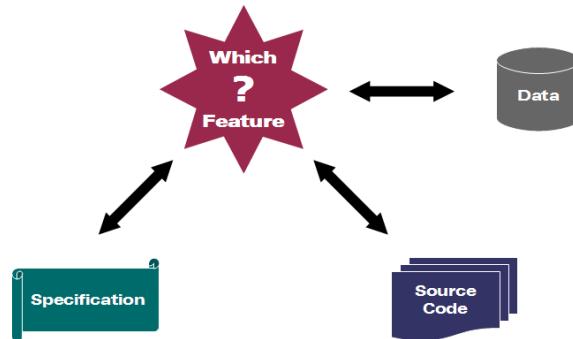


For very large models, an ABSE Tree can be partially loaded, thus allowing for manipulation on systems with limited resources.

Traceability

Two ABSE features provide good support for traceability: Atom composition, and link parameters.

Composition is the act of building a larger object (or system) from smaller parts. You can compose larger Atom Templates by reusing a putting other Atom Templates together in an orchestrated way. When you compose an Atom Template, you are automatically creating an implicit forward-trace and back-trace mechanism.



Requirements traceability is concerned with documenting the life of a requirement and to provide bi-directional traceability between various associated requirements. It enables users to find the origin of each requirement and track every change which was made to this requirement.

The same is valid for features. When we implement a certain system feature, it is important to know, at any project stage, what was coded to support such feature. The opposite is also important: to know to which feature a specific portion of code belongs to.

Link parameters are Atom Template parameters whose values are references to other Atoms. For instance, if you have a requirements Atom, you can, when you implement the corresponding system feature, have a link back to the original requirement. This gives you two possibilities: to know to which requirement a system feature is fulfilling and, conversely, know which system parts are fulfilling that particular requirement.

PART II : AtomWeaver Introduction

About AtomWeaver

AtomWeaver is an Integrated Development Environment (IDE) that implements the ABSE development automation framework.

If you are currently using another IDE, AtomWeaver is not meant to replace it. Instead, AtomWeaver works in conjunction with your current tool set, speeding up the development of your source code. You will continue to use your favorite IDE and tools to compile, test and debug.

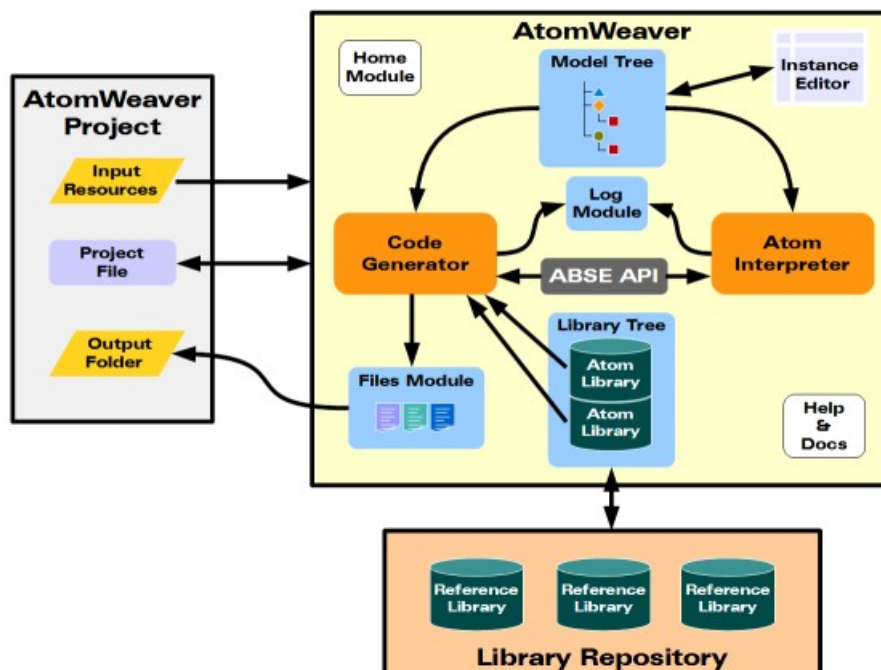
But AtomWeaver is not a "fire and forget" code generator. It is meant to support your application's complete life-cycle. From the moment you create a model, until your application life ends, AtomWeaver can support its evolution: defining requirements, generating the first code, making changes to the model, refactoring code and Atoms, managing issues, etc.

By implementing ABSE, AtomWeaver has some distinguishing factors over a regular IDE:

- It helps you cut repetitive coding, automatically decreasing maintenance costs, since the resulting code base is smaller: there's no need to maintain several instances of the same code.
- The abstraction level is higher: You can work mostly at the concept level and not at the code level.

AtomWeaver architecture

The following diagram gives you a quick overview of AtomWeaver's architecture:



Its two most important sub-systems are the Atom Interpreter and the Code Generator. The Atom Interpreter helps you build your project's model, along with the Instance Editor. The Instance Editor builds automatic editors for your Atom Instances.

AtomWeaver keeps a project as a folder that contains, besides its project file (".awp"), one folder for input resources, and another for generation results. Your Atom trees are stored on the project file.

The ABSE API is also at the core. Your reusable assets (Atom Templates), stored and organized into Atom Libraries, rely on it to generate code. The Code Generator loads your Model Tree, which drives the execution of your Atom Templates. By calling the ABSE API, generated artifacts are obtained. These artifacts are kept on the Files Module, and then saved by your command to the project's Output Folder.

The Atom Interpreter and Code Generator (as well as other sub-systems) send their messages to the Log module, that can be checked for interpretation or generation problems.

Finally, the Library Repository is a separate storage location where you can save your Atom Libraries for later reuse on other projects.

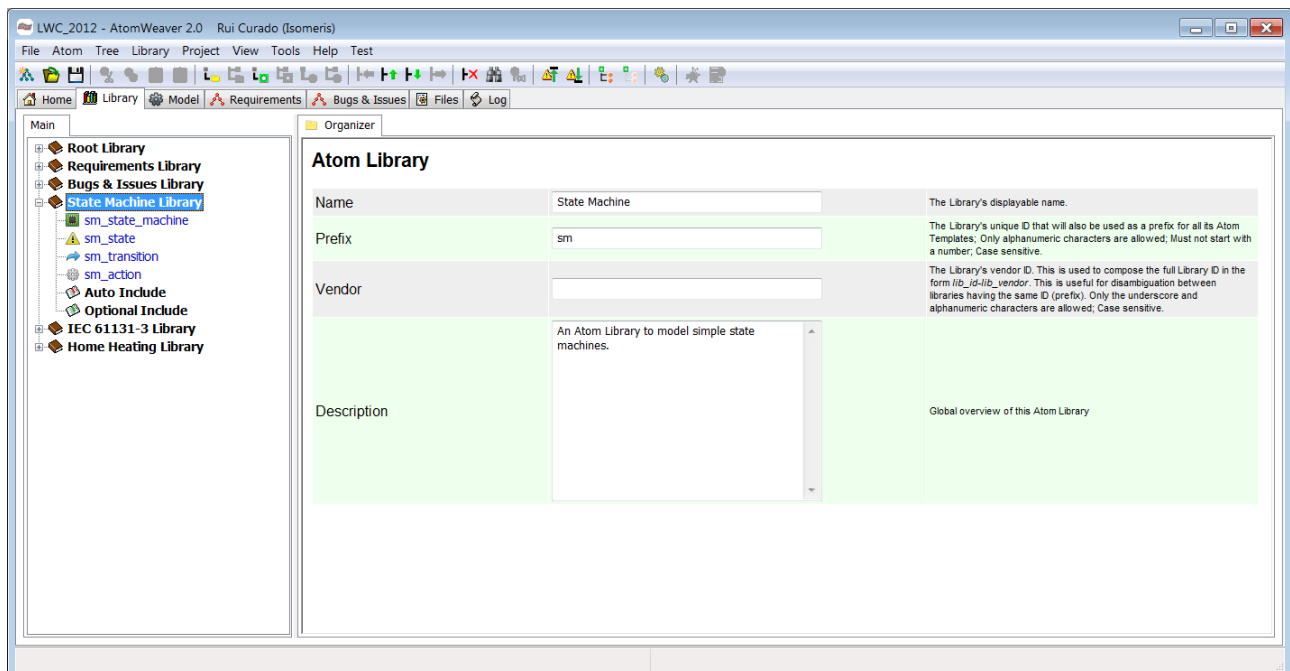
PART III : LWC 2012 Assignment

The objective of this assignment is to allow P&I (piping & instrumentation) domain experts to build a model of a home heating system.

A PLC unit will control the integrated system. For this assignment, the TwinCAT PLC simulator will replace the real hardware.

The State Machine Atom Library

Because some of the components of the system can be modeled through state machines, we'll create an Atom Library to model state machines:



Simple state machines can be modeled by creating just four Atom Templates:

sm_state_machine : Defines a state machine
 sm_state : Defines a machine's state
 sm_transition : Defines a state transition according to an input condition
 sm_action : Defines an action to perform during a state transition

By creating these four Atom Templates, we are now able to model simple state machines.

These four Atom Templates can be instantiated according to some predefined rules. These rules and other attributes are set on the Atom Template's Admin Section. Let's look at the sm_transition Atom Template:

```

Admin Form Create Pre Exec Post Functions
icon("arrow-curve")
under("sm_state")
label("@condition")

param_text("def","condition","Condition","Condition that must be fulfilled to enable the transition");
param_link("def","target_state","Target State","states","State to reach if condition is met")

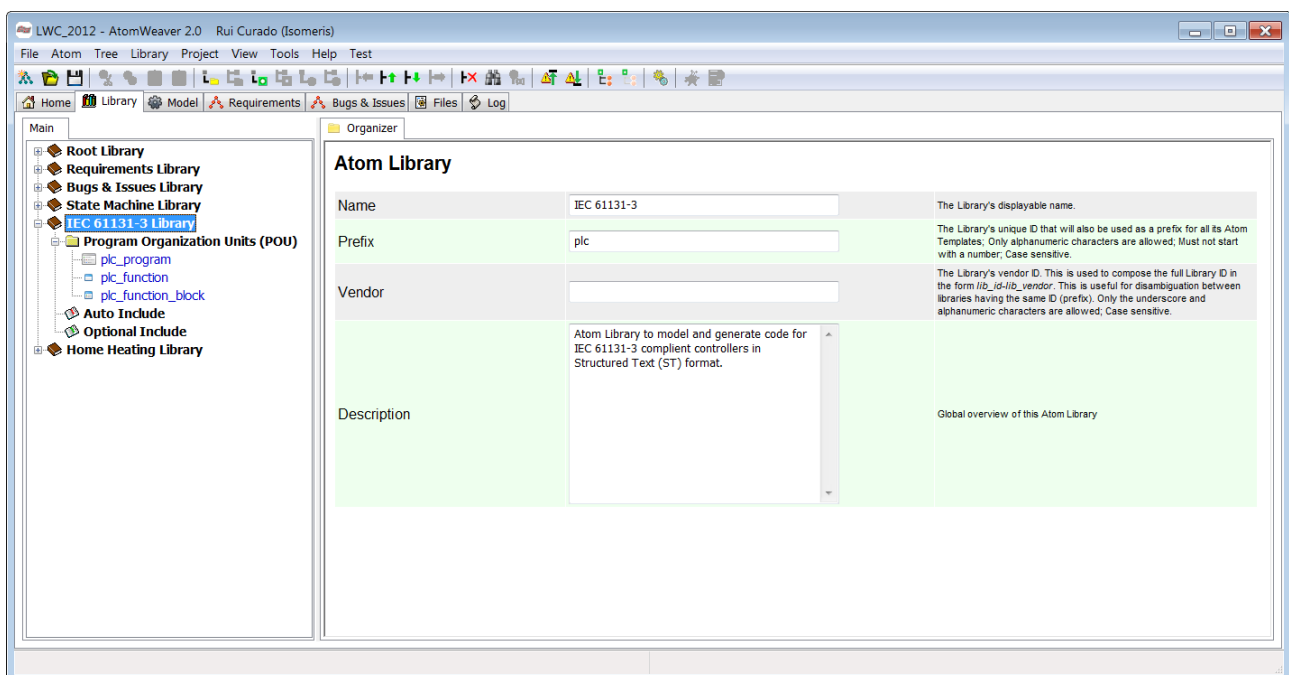
```

The `icon()` and `label()` commands set presentation attributes for all Instances of this Template. The `under()` command defines that only an Instance of `sm_state` can be a parent of an Instance of `sm_transition` (this Template). This simple but precise method ensures that every state machine transition you create is within context of its input state.

The `param_text()` and `param_link()` commands define the Atom Template's variability. For each variation point, a parameter is defined. When instantiating this Template, AtomWeaver will know how to build an editor based on this information.

The IEC 61131-3 Atom Library

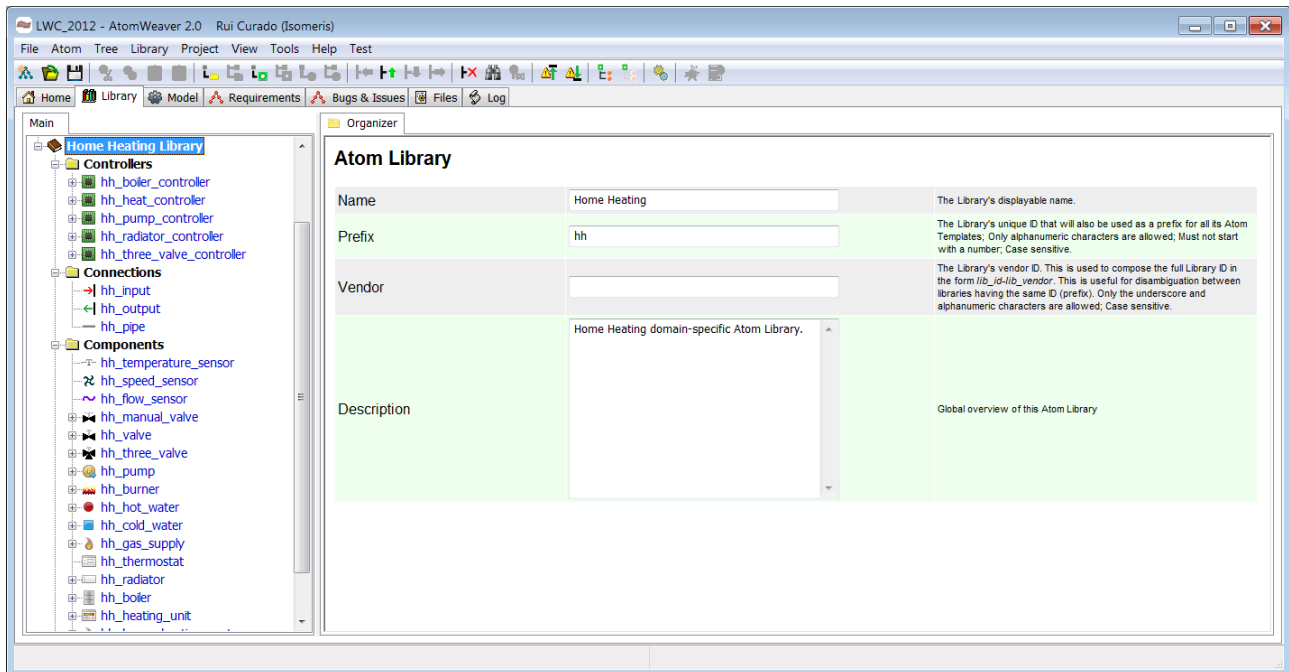
Because we're going to write some programs in IEC 61131-3 compliant format, we'll create an Atom Library that models the necessary subset of features we need for the assignment.



For this assignment we'll just create three Atom Templates that model the IEC standard's higher-level Program Organization Units (POUs).

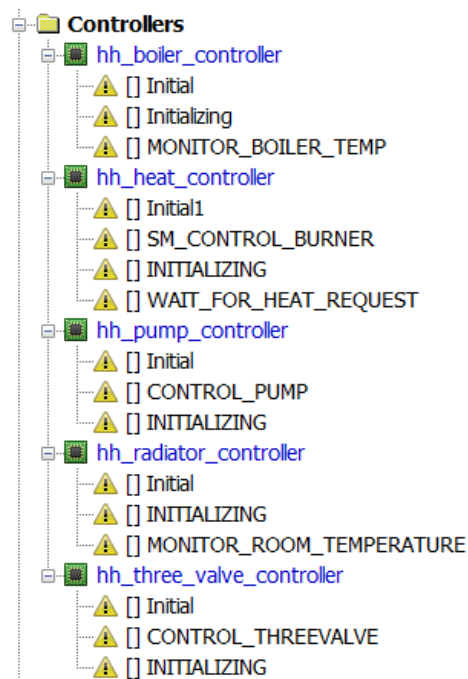
The Home Heating Atom Library

The main Atom Library of our assignment is the Home Heating Library, where the higher-level, domain-specific Atom Templates are defined:



Controllers

The controllers of some active components are modeled on this Atom Library. Because these controllers are state machines, we can model a controller by inheriting from `sm_state_machine`. Then, we can reuse the other state machine Atom Templates to model specific controllers.



These metamodels can now be reused on higher-level constructions. In our case, we'll apply them on the home heating system components.

Connections

Three Atom Templates help the domain expert connect the components that make up the home heating system:

`hh_input` : Defines an input point in a component.

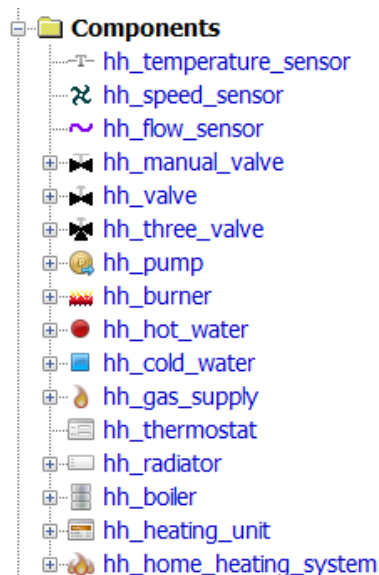
`hh_output` : Defines an output point in a component.

`hh_pipe` : Connects an output point of a component to an input point of another.

We will instantiate these Atom Templates to define a component's input and output connections, as well as their inter-connection.

Components

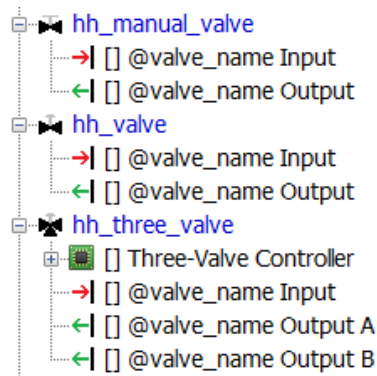
All the components that are necessary to model a home heating system are included here:



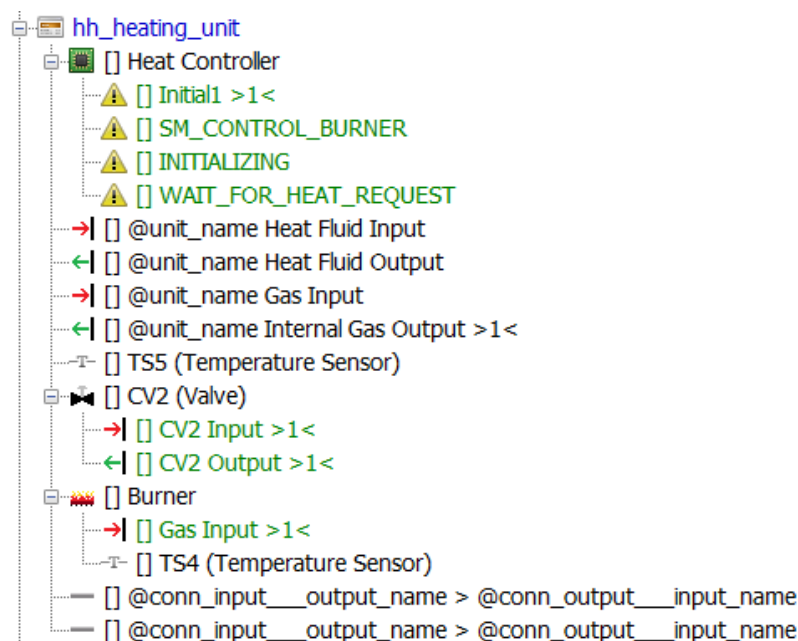
Sensors, valves, small components (pump, burner), system input/output (gas, hot and cold water supplies) and major components (thermostat, radiator, boiler and heating unit), by means of Atom Templates, were turned into reusable assets that can then be easily instantiated by the domain expert on any home heating model.

Solution Composition

Higher-level concepts are built by reusing other lower-level concepts. For instance, the valve metamodels were completed by composing the necessary inputs and outputs. In addition, the `hh_three_valve` metamodel reuses the definition of a three-valve controller state machine through the instantiation of `hh_three_valve_controller`.



The heating unit is a composite component because it internally contains other components. Following the established metamodel rules, we can model the heating unit internals:

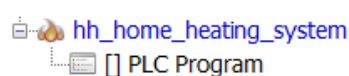


Here we can see the powerful combination of inheritance and composition to create a reusable asset that also reuses other assets. This helps you follow the DRY (*Don't Repeat Yourself*) principle, reducing the maintenance footprint of your development.

Green-labeled Atom Instances are *Auto-Generated Instance Slaves*: Atoms that are being automatically reused and managed by AtomWeaver. These Atoms are connected to their *masters*. Whenever their master changes, they get changed too.

The Home Heating System metamodel

There is one top-level Atom Template that is used to start the modeling process:



Since we are targeting the TwinCAT simulation environment, this project will generate a TwinCAT export file (".exp extension). Because hh_home_heating_system will be the root Instance of the

whole project model, we know that it will be the first Atom to generate code. Therefore, this Atom should generate the export file's architectural elements.

A closer inspection reveals that TwinCAT's project elements are grouped into categories: On an export file, the simulation POU's come first, then the project POU's come next, then infrastructure. The main program, data types, visualization and resources then follow.

Based on this, `hh_home_heating_system` will contain this generation code:



```

root_default()

cursor("begin", "@output_file")

section("simulation")
section("project")
section("infrastructure")
section("main_program")
section("data_types")
section("visualization")
section("resources")

```

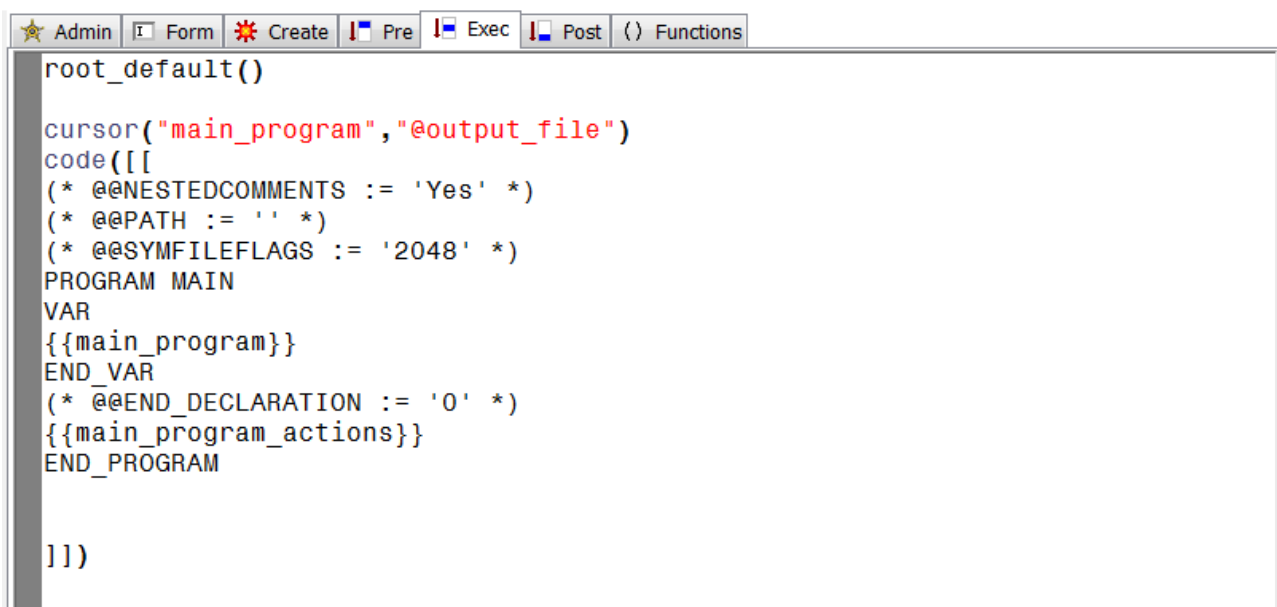
This Atom will create the export file, and then define seven file sections that will receive the corresponding code that will be generated next by other specialized Atoms. For example, to generate code for the simulation, any other Atom just has to include a command like

```
cursor("simulation", "@output_file")
```

to redirect the Generation Cursor to the appropriate place.

Then, we chose to add the main PLC program to the home heating system metamodel as this corresponds to the MAIN POU in the PLC. To do this, we instantiated the `plc_program` Atom Template under the `hh_home_heating_system` Template. This way we ensure that every home heating project will contain the MAIN POU.

Because of this, the `plc_program` Atom Template has the following generation code:



```

root_default()

cursor("main_program", "@output_file")
code([[
(* @@NESTEDCOMMENTS := 'Yes' *)
(* @@PATH := '' *)
(* @@SYMFILEREFLAGS := '2048' *)
PROGRAM MAIN
VAR
{{main_program}}
END_VAR
(* @@END_DECLARATION := '0' *)
{{main_program_actions}}
END_PROGRAM

]])

```

With this block of Lua code, we transform the PLC main program definition into final source code.

We move the Generation Cursor to the `main_program` section in the current output file, and then generate a block of source code. In the generated code, we mark two places to receive additional code:

```
{{main_program}}
{{main_program_actions}}
```

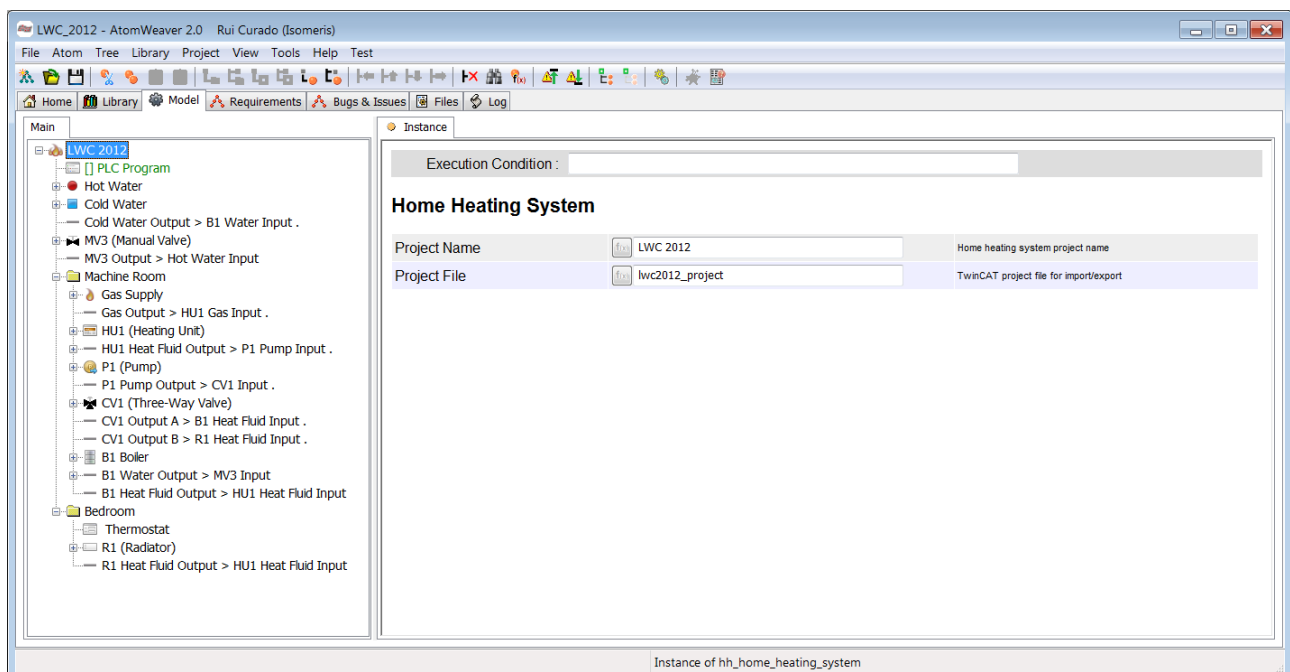
These are called *embedded file section definitions*. Because code generation in ABSE is a cooperative effort between several Atoms, you can mark specific places in your code that can be accessed by other Atoms and receive their generated code.

In this specific situation, these two file sections will later contain code generated by some of the home heating system components.

Modeling the Home Heating System

Modeling the home heating system is now quick and straightforward. Thanks to ABSE's reuse mechanisms and construction constraints, the domain expert only needs to pick the necessary components from a list, and connect them using the appropriate model.

The complete home heating system from the assignment can be seen below:

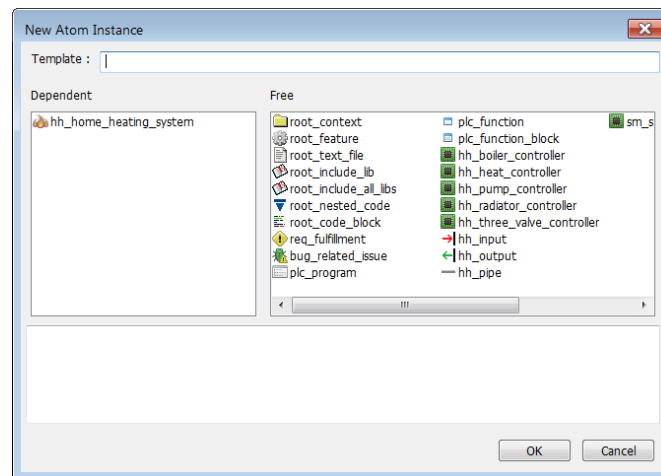


AtomWeaver's model editor is split in two parts. The left side shows the ABSE model tree, and the right side shows the automatic editor for the currently selected Atom Instance.

Starting out

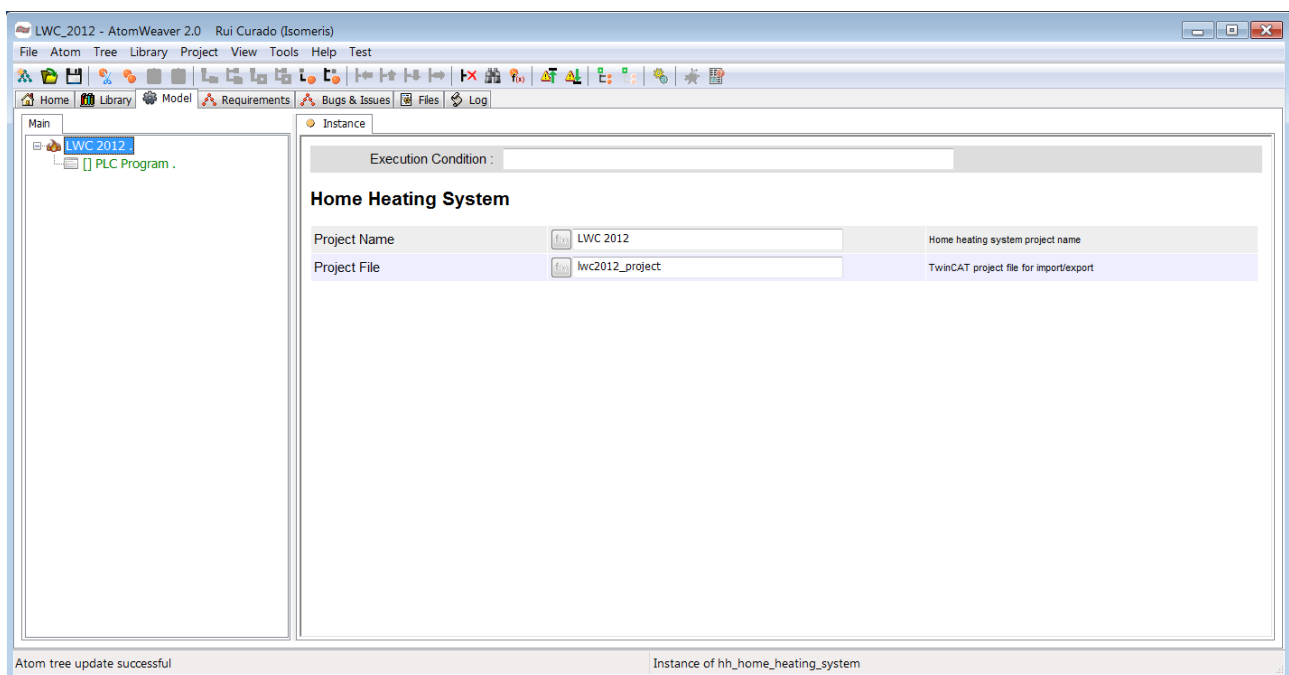
The domain expert starts with an empty model. If the construction constraint are correctly defined,

AtomWeaver will suggest what Atoms should be instantiated according the current selection. AtomWeaver suggests the first Atom to instantiate:

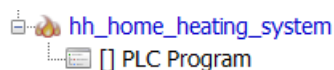


The suggested Atoms are shown on the left-side list. The other list shows *Free Templates*: those that do not have any construction constraints applied to them. To avoid confusion during the model construction phase, the number of Free Templates should be kept to a minimum.

After selecting the `hh_home_heating_system` Atom Template, AtomWeaver instantiates in on the model tree, and presents its automatic editor.



Notice the automatic creation of an additional Atom Instance (*PLC Program*). Checking the `hh_home_heating_system` Atom Template, we can confirm the presence of an *Atom Instance Master*:

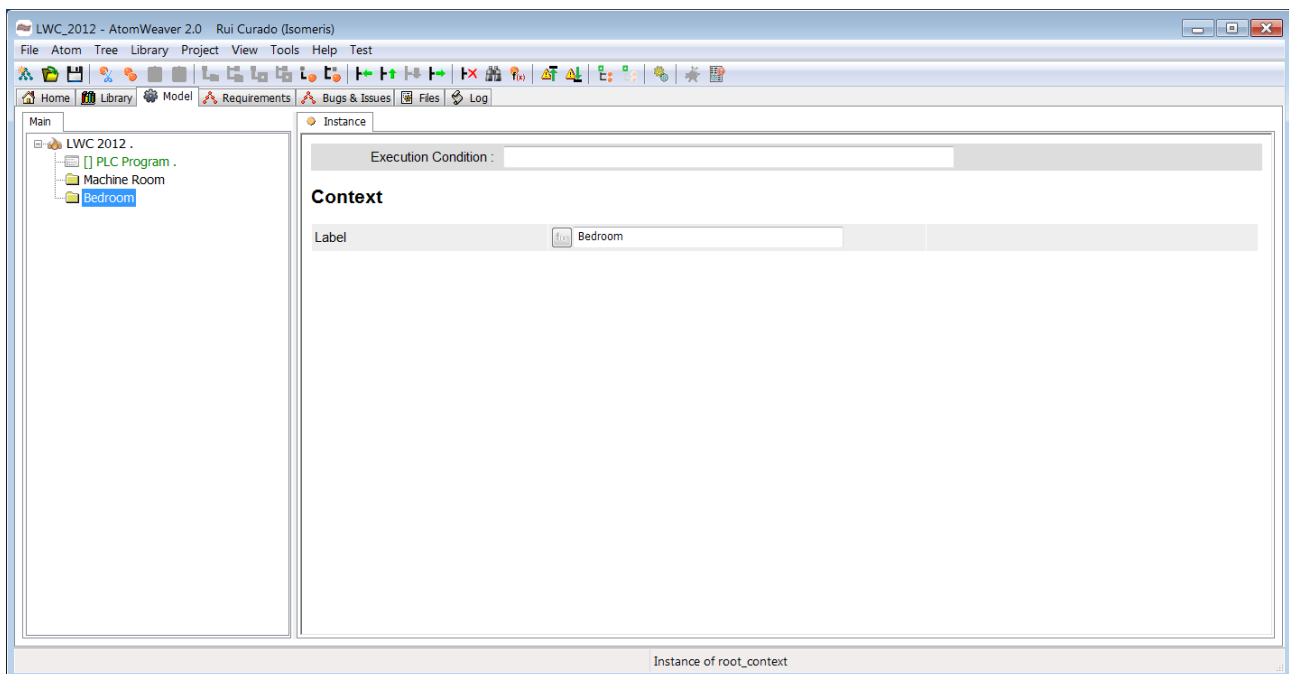


ABSE commands that all Atom Instances under an Atom Template are automatically duplicated for any instantiation of that Template. In addition, the duplicated Instances become slaves of their master instances: Any changes to the Template or any of its Master Instances are automatically propagated.

Separating the installation into physical spaces

While not a requirement on the LWC assignment, we decided to separate the home heating system into physical spaces, to help organize the resulting model.

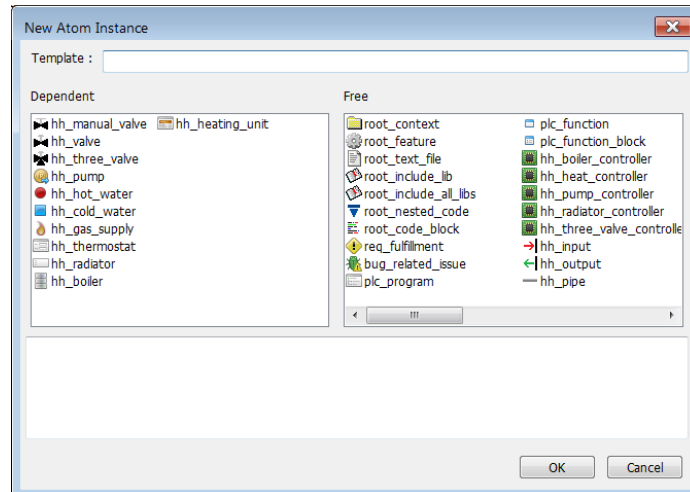
For that matter we created two instances of `root_context`. This predefined Atom Template has the *neutral* property: it is invisible to construction constraints. Atoms instantiated from this Template will be made neutral. We can use them to organize our model without breaking the established home heating system modeling rules.



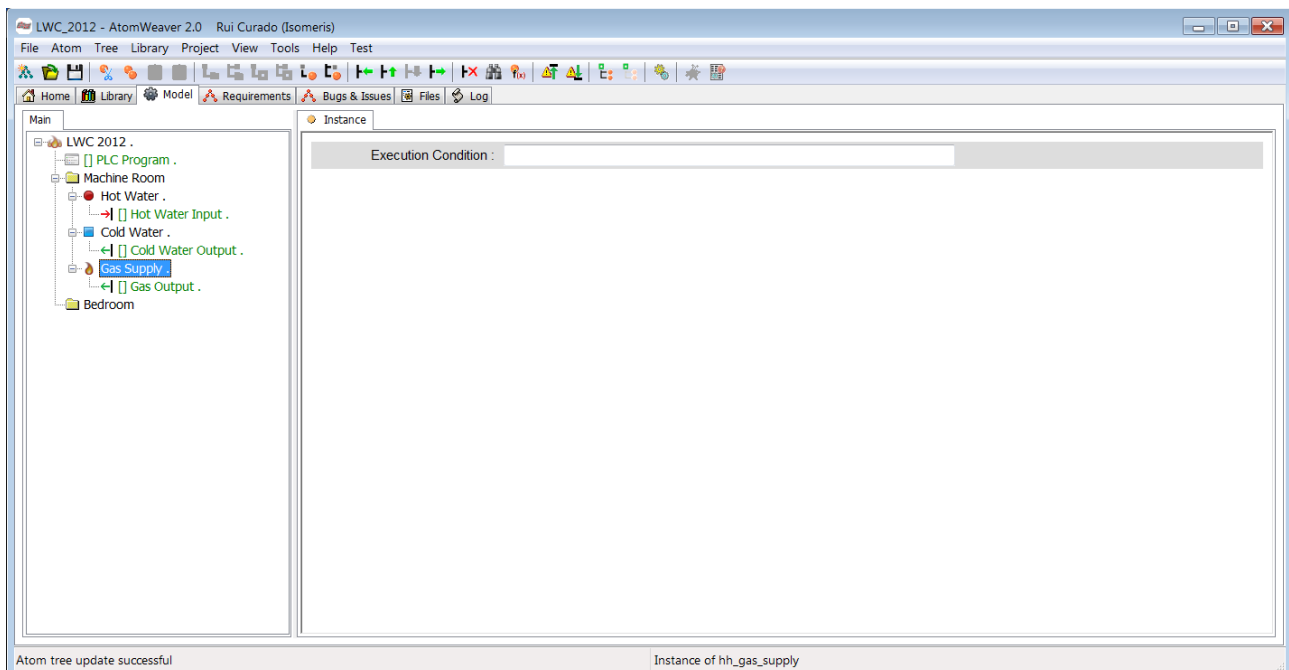
Creating the water/gas supply points

Now that we have identified the places where the heating system will be installed, we'll create the primary external exchange points (cold water in, hot water out, gas in). This could be further automated and be automatically created by the `hh_home_heating_system` Atom Template, but we would lose some flexibility.

Selecting the Machine Room as the creation context, we now create the exchange points:



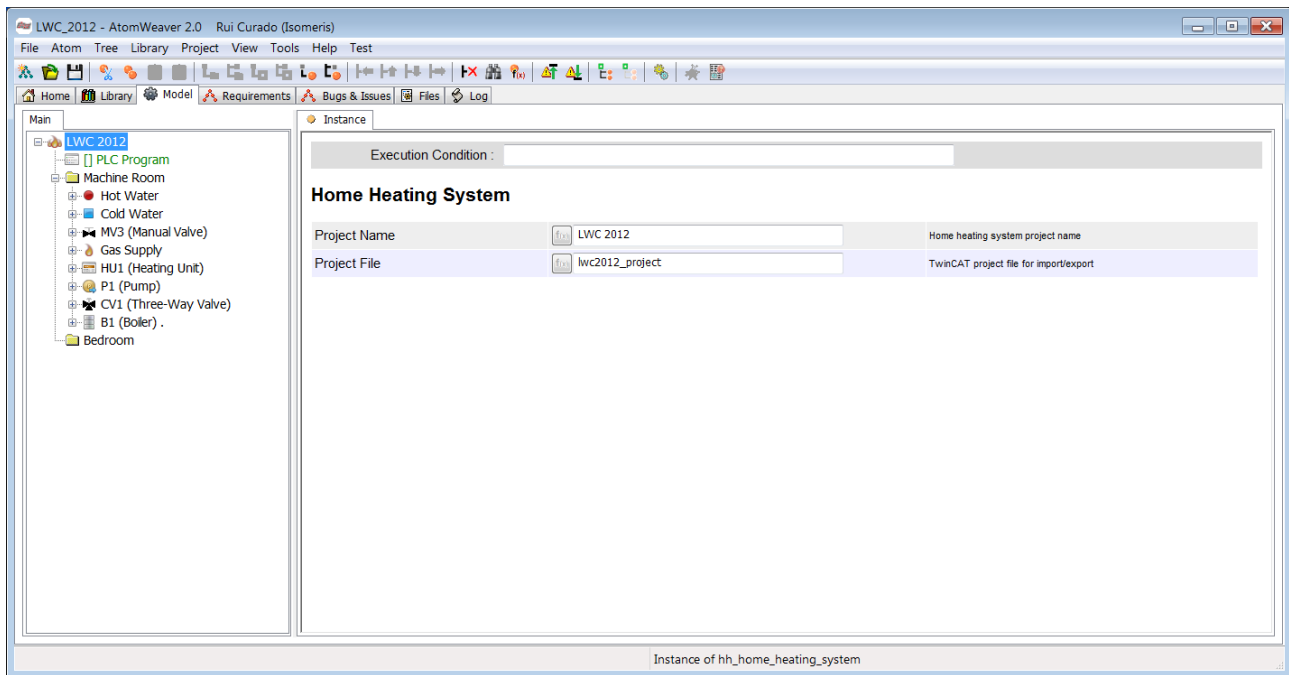
Notice that, because the current creation context is a neutral Atom (*Machine Room*), the construction constraints of its parent Atom (*LWC 2012*) are instead considered.



As before, Atom Instances defining the connection points of the external exchange points were automatically duplicated as result of the instantiation of `hh_gas_supply`, `hh_hot_water` and `hh_cold_water`.

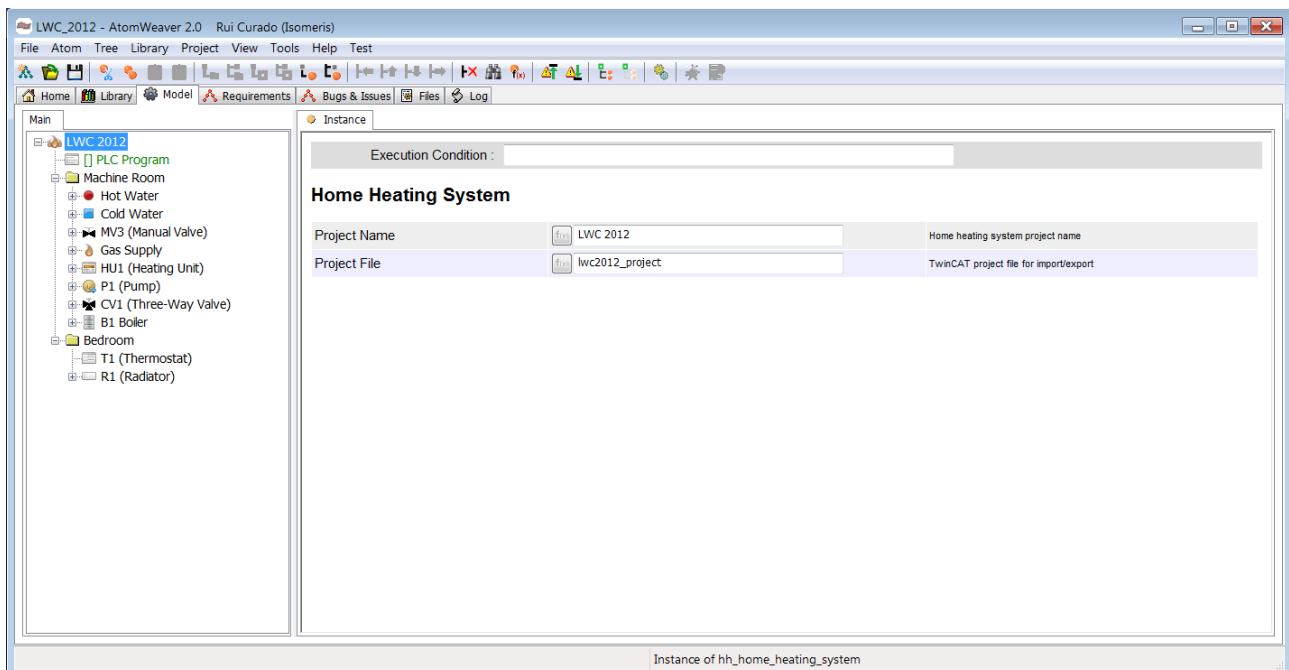
Creating the rest of the Machine Room's components

We will now create the other necessary components located on the *Machine Room*: A manual valve, the heating unit, the boiler, a pump and a three-way valve. The creation process is similar to we have seen so far.



Creating the Bedroom's components

The bedroom will have a radiator and a thermostat installed. We'll now create these components:



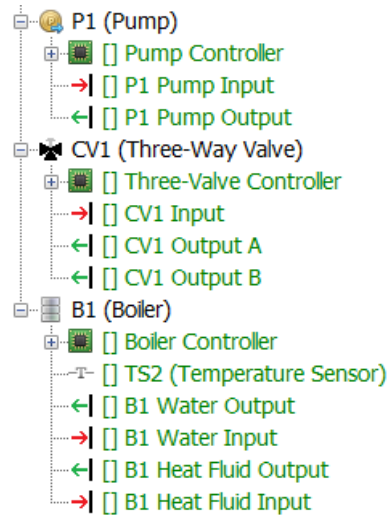
We have now created almost all of the components that are necessary for our system. Some sensors are still missing and we'll place them later on.

Connecting two components

Now that we have the necessary components, we'll complete the system by establishing the necessary water and heat fluid circuits by connecting the system's components with a specialized

Atom: hh_pipe.

To make these connections possible, we have decided that each component has already specified what are its connection points, and if they are inputs or outputs.



Above you can see a partial view of the heating system's Atoms we have just created (those with black labels, the others were automatically created). Automatically created were the definitions of their connection points.

We will now add a pipe to connect the *Cold Water* output to the boiler's water input.

The screenshot shows the AtomWeaver 2.0 interface with the 'hh_pipe' atom selected. The left pane shows a project tree with 'LWC 2012' as the root, containing 'PLC Program', 'Machine Room', 'Hot Water', 'Cold Water', and several components like 'MV3 (Manual Valve)', 'Gas Supply', 'HU1 (Heating Unit)', 'P1 (Pump)', 'CV1 (Three-Way Valve)', 'B1 Boiler', 'T1 (Thermostat)', and 'R1 (Radiator)'. The right pane shows the configuration form for the 'hh_pipe' atom, which includes an 'Execution Condition' field, an 'Attributes' section with 'Diameter' and 'Length' fields, a 'Connection' section with 'Input (From)' and 'Output (To)' fields, and a 'BOM' section with an 'Internal Pipe' checkbox.

The `hh_pipe`'s automatic form asks for some information about the pipe to be used on the connection, and asks you to select the end points. Finally, it asks if the pipe is already included in a component. If it is, then it won't be billed separately.

To connect two components through this pipe, you'll use the *Select From List* button on the Atom's editor form:

Instance

Execution Condition :

Attributes

Diameter	<input type="text"/>	Pipe diameter
Length	<input type="text"/>	Pipe length

Connection

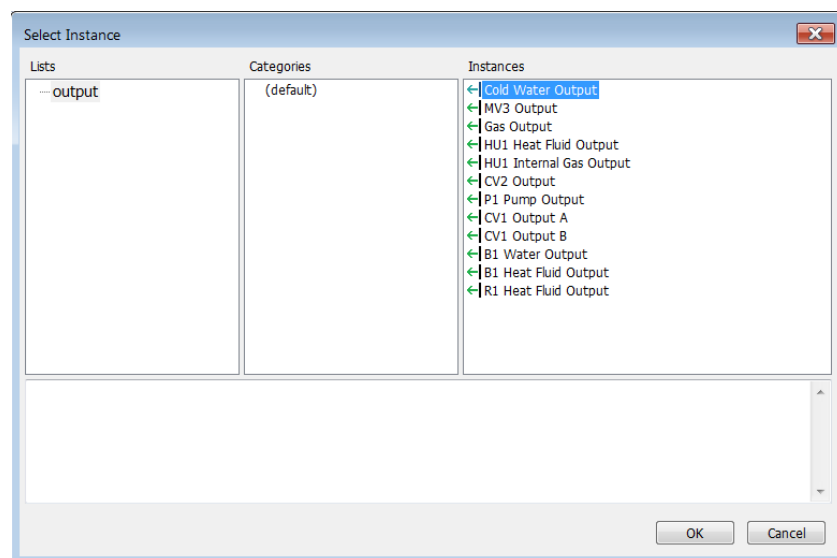
Input (From)	<input type="text"/>	Input point of this connection
Output (To)	<input type="text"/>	Output point of this connection

BOM

Bill of materials

Internal Pipe	<input type="checkbox"/>	This pipe is internal to an equipment and is not to be billed
---------------	--------------------------	---

The Atom List Selection dialog is shown. Because the input of the pipe can only connect to the output of a component, only the available outputs are shown:



AtomWeaver has adapted itself to the needs of the domain expert, making his job easier. To make this productivity helper possible, only two commands were needed:

```
list("output")
```

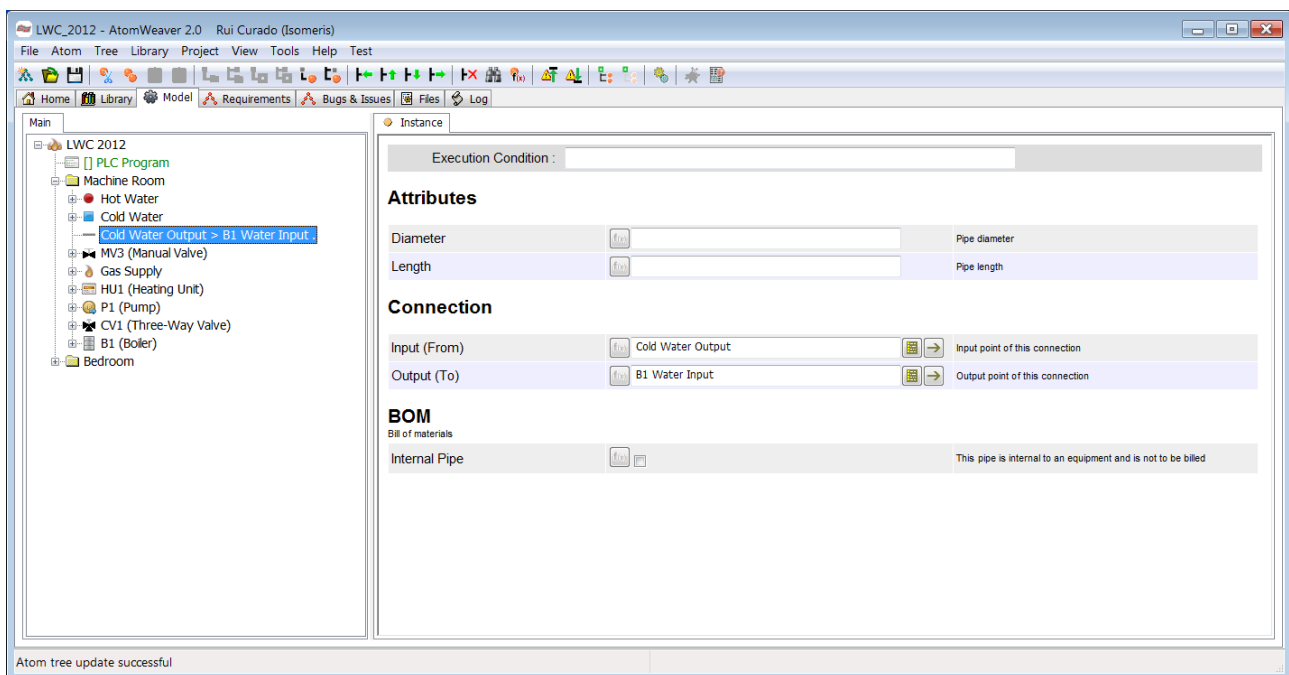
on hh_output's Create Section, and

```
param_link("def", "conn_input", "Input (From)", "output", "Input point of  
this connection")
```

on hh_pipe's Admin Section. The `list` command registers each component output on a global Atom List named *output*, and the `param_link` command specifies a link-type parameter for the pipe metamodel that will only accept Atoms from the *output* list.

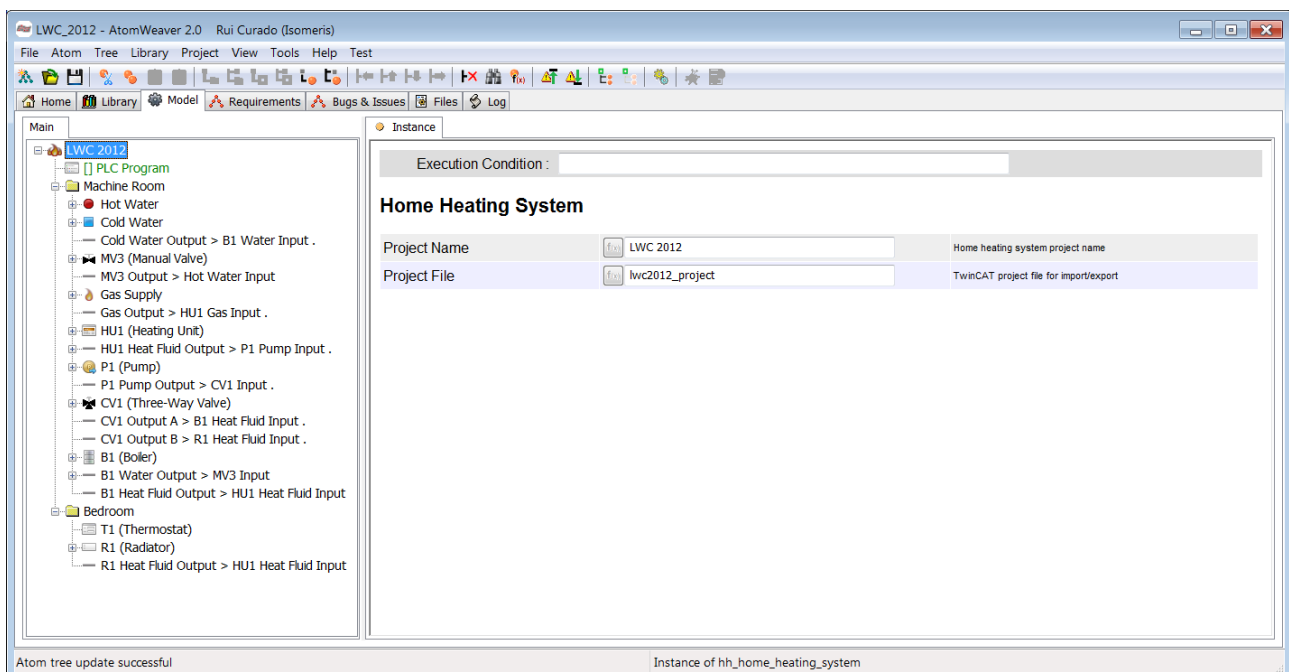
An Atom List is a mechanism to categorize your project's Atoms and make their selection easier.

We can now do the same thing to the other connection. The Atom's label identifies the connection:



Completing the system's circuits

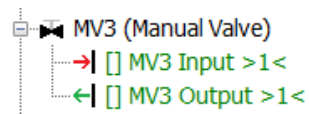
We can now repeat this operation to connect all the components together. Because `hh_pipe` has to be flexible, and because it connects two different components, there is no unambiguous solution to their placement in the tree. We have therefore decided that each pipe is created next to its input component:



Checking connections

To easily check if all component inputs and outputs are connected we can look at the input/output

labels:



The $>n<$ suffix at the end of an Atom's label indicates the number of Atoms that are referencing it. In this case, if an input or output has at least one referencing Atom, it means a pipe is connected there.

You can also jump to the pipe's end points using the *Jump To Atom* button on the editor's form:

Connection

Input (From)	MV3 Output		Input point of this connection
Output (To)	Hot Water Input		Output point of this connection

Generating Code

At any time, the Code Generator can be invoked to obtain the PLC code for the execution and simulation of the home heating system on TwinCAT. The results are presented on the Files Module.

Conclusions

After finishing the LWC assignment, we can take some conclusions:

Reuse in action

We could witness the productivity gains of applying automation and reuse when modeling a home heating system using the ABSE framework: We had to create 26 Atom Instances on the model, but at the end we had a total of 78 Atoms. This is because the other 52 were automatically created by Atom inheritance and composition mechanisms.

Model Tree

Atom Instances : 78
Auto-Generated Atom Instances : 52
Relocated Atom Instances : 0
Used Memory : 0.1 Mb

This roughly translates to 66% savings in effort as opposed to regular modeling. Savings regarding manual coding are not calculated but would easily reach 90%.