An open toolchain for generating Modelica code from Building Information Models

Matthis Thorade[1] Jörg Rädler[1] Peter Remmen[2] Tobias Maile[3] Reinhard Wimmer[3] Jun Cao[3]
Moritz Lauster[2] Christoph Nytsch-Geusen[1] Dirk Müller[2] Christoph van Treeck[3]

[1]Berlin University of the Arts (UdK), {m.thorade,jraedler,nytsch}@udk-berlin.de
[2]RWTH Aachen University, E.ON Energy Research Center, Institute for Energy Efficient Buildings and Indoor Climate, Aachen, Germany {premmen,mlauster,dmueller}@eonerc.rwth-aachen.de
[3]RWTH Aachen University, Institute of Energy Efficient Building, Aachen, Germany
{maile,wimmer,cao,treeck}@e3d.rwth-aachen.de

Abstract

Building Performance Simulation (BPS) is a key element in the design of energy efficient buildings, and there is increasing interest in using the Modelica modelling language for BPS. The IEA-EBC coordinates development of BPS in Modelica in the project "Computational Tools for Building and Community Energy Systems" (Annex 60). However, developing BPS models and collecting required input data are time-consuming and error-prone processes. Reusing existing Building Information Models (BIM) as basis for Building Performance Simulation (BPS) has the potential to make BPS model development and subsequent simulation easier, faster and more reliable. Activity 1.3 of the Annex 60 project is working on an open-source toolchain that can semi-automatically generate code for BPS Modelica models from a BIM data source. Parts of that toolchain are discussed in this paper.
Keywords: Building Information Modelling, Modelica code generation, Building Performance Simulation

1 Introduction

Buildings become increasingly integrated to reduce energy and peak power and to increase occupant health and productivity, leading to complex building design. Building Performance Simulation (BPS) is one key element in the design of energy efficient buildings. The Energy in Buildings and Communities Programme (EBC) of the International Energy Agency (IEA) launched in 2012 the project "Computational Tools for Building and Community Energy Systems", also know as Annex 60 (Wetter and van Treeck, 2012). The Annex 60 project aims at developing next generation computing tools for the buildings industry, based on open non-proprietary standards, including the Modelica modelling language and the Functional Mockup Interface. The structure and organization of the project into subtasks and activities is shown in Figure 1.

The development of Modelica model libraries for BPS before Annex 60 was fragmented with the result that each institution was developing the same components in a different, possibly incompatible, manner. Activity 1.1 focuses on harmonizing BPS library development by providing a core library of base classes and components commonly needed. The libraries currently contributing to and relying on the Annex 60 core library are:

- AixLib from RWTH Aachen (Fuchs et al., 2015)
- BuildingSystems from UdK Berlin (Nytsch-Geusen et al., 2013)
- Buildings from LBNL (Wetter et al., 2014)
- OpenIDEAS from KU Leuven (Baetens et al., 2015)

Each library extends the core library by providing additional components for special applications, depending on the respective institutions research focus. The libraries AixLib and BuildingSystems will be used later in this paper to demonstrate the code generation.

But even with advanced component libraries available, building up BPS models from hand and collecting required input data remain time-consuming and error-prone processes (Bazjanac et al., 2011), preventing practitioners from using BPS more extensively in standard planning processes. Building Information Modelling (BIM) is a well established technology to model and manage the digital representation of a building over its entire lifecycle (see e.g. Eastman et al., 2008). Reusing existing Building Information Models (BIM) as the basis for Building Performance Simulation (BPS) has the potential to make BPS model development and subsequent simulation easier, faster and more reliable.

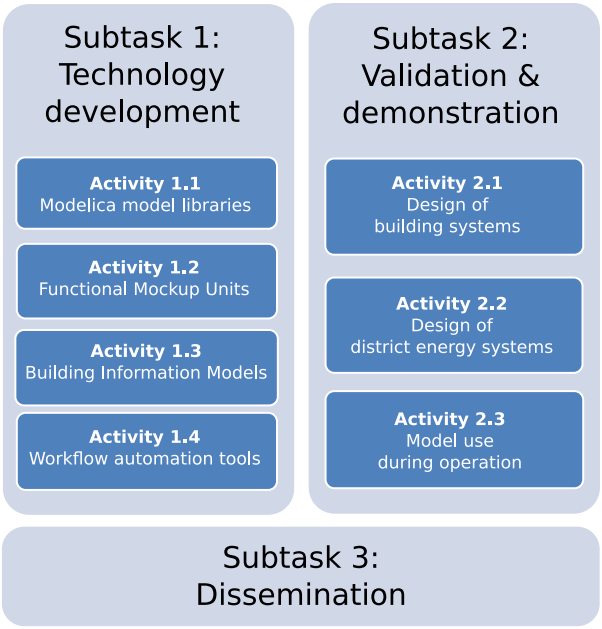

Figure 1. Structure and organization of the IEA EBC Annex 60 (figure adapted from Wetter and van Treeck (2012)). In this paper, results from Activity 1.3 are presented.

Activity 1.3 of the Annex 60 project aims at leveraging such BIM systems as a source of information for semi-automated generation of BPS models in the form of Modelica code. In order to reach a broad audience, the methods and tools developed should:

- make use of, comply with and contribute to existing open standards,
- support both building geometry as well as heating, ventilation and air conditioning (HVAC) components,
- support multiple Modelica libraries,
- support a high degree of automation,
- be a collection of small, reusable tools,
- be released under open-source licenses.

A description of the toolchain implementing the overall process as planned in this project is given by Remmen et al. (2015), including a review of prior work in the field. That paper is also summarized in the following section, giving an overview of the overall process, basic ideas, assumptions and software foundation. The section Python Framework then describes in more detail the part of the toolchain that is used for controlling the workflow, via a GUI or via Python scripts, as well as the actual code generation. The section Use Cases gives a first, simplified demonstration of the process. The paper then concludes with a short discussion of limitations and future work.

2 Process Overview

The whole process of generating Modelica code from Building Information Models is in this project implemented as a toolchain of various special-purpose tools. Having various tools with a clearly defined task means these tools can be developed partly independent, and each block can possibly be reused in a different context. On the other hand, interfaces between the tools have to be clearly defined, either in the form of a file format or as an Application Programming Interface (API). The various steps of the process and the interfaces are shown schematically in Figure 2 and are discussed in the following paragraphs, summarizing the paper by Remmen et al. (2015).

Creating the Building Information Model A typical starting point for a BIM-based workflow would be an architect creating a designated space and usage structures using a BIM-based CAD software. Other domain experts, e.g. HVAC engineers, then enrich the BIM by contributing further data. In order to collaboratively create the model, all involved actors use a common file format. IFC is a well-established, non-proprietary and standardized BIM file format (International Organization for Standardization, 2013). In this project, we rely on version 4 of IFC, because it contains several improvements over its predecessor IFC 2x3. Using a standard format means that various applications on the market will be able to deliver input to our toolchain. Also, we profit from existing tools for checking, viewing or sanitizing IFC files.

Transformation to Simulation Domain Model While IFC is a well-established BIM file format, it is in its current form not very well suited as direct input for Building Performance Simulation (BPS), because it does not contain all information required for BPS (e. g., some detailed HVAC objects and properties are missing, as well as simulation specific objects and properties), and it has rather long turnaround time for changes. Information models for the simulation domain and corresponding file formats have been developed with the goal to resolve these drawbacks. In this project, we rely on SimModel and corresponding SimXML files as defined by O'Donnell et al. (2011). The SimXML file format is clearly defined by an XML Schema Definition (XSD). SimModel closely aligns with IFC regarding building geometry and building physics, but removes some redundancies and simplifies relationships between objects. Besides the IFC model it also entails other datamodels such as gbXML and others. Regarding HVAC components, SimModel is a superset of IFC. That means missing information has to be added during the transformation process. For conversion of geometry and building physics data ex-

DOI 10.3384/ecp15118383

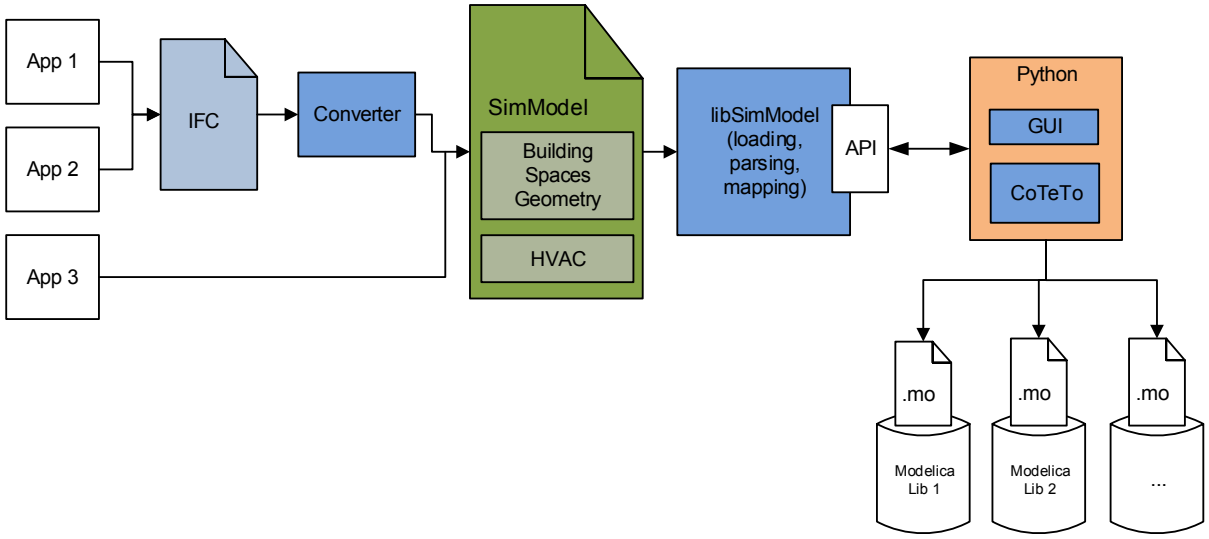

Figure 2. EnEff-BIM Process Overview.

isting tools can be used, such as Simergy and/or Space Boundary Tool (SBT) (Rose and Bazjanac, 2015). For conversion of HVAC data a new tool is developed in this project. The two converted parts have to be combined in a second step into a single valid and correct SimXML file that is information complete, as expected by the following steps in the overall process. Another option that is also investigated in this project is to create (parts of) SimModel files with only few high level input parameters provided by the user (such as year built, type of building, etc.) and filled with typical and statistical data for a complete model.

Mapping to Modelica libraries To accomplish the link between the rigid data structure in SimModel with the flexible data representation on the Modelica side, mapping rules are needed (Wimmer et al., 2014). Loading the SimModel file, parsing it and mapping the data from SimModel to Modelica is performed by the C++ library libSimModel, described in detail by Cao et al. (2014, 2015). The SimXML file is first loaded by a validating parser that uses the XSD. As part of the parsing process, a hierarchical tree is built up and some manipulations and simplifications, like resolving links, are performed. The data, once loaded, is then mapped using the library specific mapping rules as described by Wimmer et al. (2015). These mapping rules are valid for a specific version of a specific library. When the library changes, the corresponding mapping rules have to be updated. The mapping rules are again stored in XML files (confirming to a corresponding XSD). To ease maintenance of the mapping rules, a tool for conversion between a spreadsheet table and the corresponding mapping rule XML file is developed.

All mapped data and, as needed, also unmapped data, as well as the methods and functions of the libSimModel library for loading, parsing and mapping are exposed to Python as an API.

Code Generation and User Interface The last part of the toolchain is written in Python and it covers three tasks: Process control, Information Pre-Processing and the actual Modelica code generation. These tasks and the implementation are discussed in the following section.

3 Python Tools

The Python tools cover three tasks: Process control, Information Pre-Processing and the actual Modelica code generation. The organization of the tools is shown in Figure 3.

3.1 Process Control

As discussed in the previous section, the whole process is implemented by several components that build a toolchain. For the normal end-user this chain should appear as one tool, but for power-users and during development all parts should be usable standalone. To achieve this, some requirements must be fulfilled:

- a common programming platform (language, versions),
- an Application Programming Interface (API) to call the components functions from the common programming platform,

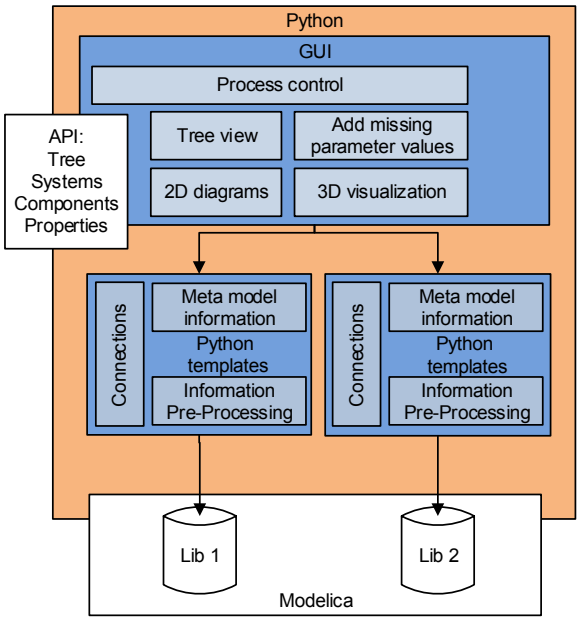

Figure 3. The Python part of our toolchain.

- no own or just an optional or partial graphical user interface (GUI).

These requirements are already fulfilled for that part of the toolchain that use SimModel as a starting point. That part will be controllable from the common process control using the GUI or scripts, standalone or embedded in other tools. Further parts of the toolchain will be added to the common process control as needed.

As the common platform for integrating the components Python is used, so all components must be usable from this language. This requirement is fulfilled if the component is already written in Python, otherwise a wrapping API is necessary. Python is well suited to bind different components to one tool, and it is widely used and accepted in the Modelica community. The GUI part is implemented in Qt, so it will be portable to all major operating systems.

3.2 Information Pre-Processing

While SimModel was designed to contain all information that is required for BPS, that information may sometimes have to be processed or converted. Simple conversions (e.g. unit conversion) will be covered by the mapping rules, but more complex conversions are more easily implemented in Python. One example for complex information conversion is the calculation of equivalent lumped heat capacities according to VDI 6007 that are used in low-order models of AixLib (German Association of Engineers, 2012).

The mapping rules also do not cover the connections between objects, this information is passed to Python as meta model information. Another task is the processing of Modelica graphical annotations. This is work in progress and will be extended as needed. The information processing is implemented as Python filters for the templates, as described in the following section.

3.3 Modelica Code Generation

The actual Modelica code generation is implemented as a tool named CoTeTo, which stands for Code Templating Tool. Although designed for this project, this tool was implemented in a way that it can be used standalone and in other software environments. CoTeTo will be released under an open-source license.

In this project, Modelica models for a set of different model libraries have to be generated using a common data source. Each library needs separate filtering and output of data because of different modelling approaches. These libraries are currently under development and are likely to change in the future as well. This requires a flexible and generic data conversion framework to allow for future changes. Thus, the framework should allow flexible output components for different libraries in multiple versions as well as flexible input components, both should be easy to maintain even for non-programmers. The workflow of CoTeTo and the coupling to other tools within the toolchain is shown in Figure 3. We designed CoTeTo to be used by graphical, command line and library level interfaces. The multiple access possibilities open the framework to a huge community. The fact that Python does not require extensive compilation cycles helps with rapid development. The following section will give an overview of the components and their functionality. We have divided CoTeTo into input components (*Data API*s) and output components (*Generators*). A *Generator* depends on a specific *Data API* (defined by its name and version).

The Template Approach There are two general concepts for the generation of textual output within a computer program. One approach is to embed `print()`-statements for text strings and data in the structure of a program. This is useful for nearly static, well-defined structures of the data set and of the textual output.

The other approach is template-based, where placeholders for the content are embedded in a text file (a template for the output). Besides placeholders templates also offer control structures. Thus, template-based model generation allows complying with fixed Modelica language syntax and adding flexible model content in the same file. One advantage is the flexibility for the end user, who does not necessarily need to dive into the programs' internal structure, but can just enrich the template file with placeholders and simple programming constructs, whenever the used Modelica models change. This workflow is much like the form letter function in office software, which fills some variable address fields in a text document from a database.

DOI 10.3384/ecp15118383

The template approach fits very well into the flexible structure of the CoTeTo framework, as it is independently usable for different information sources. From the list of available template engines Mako (Bayer, 2014) and Jinja2 (Ronacher, 2014) seem to fit best into CoTeTo. At this point support for both is implemented, but after an evaluation phase one of the engines may be dropped in the future.

Input - Data APIs A *Data API* is a Python module that defines a prescribed way to fetch data sets from a data source. Although we use the Python language to write the CoTeTo, *Data API* functions can interface to other languages.

Different *Data APIs* and different versions can be used in parallel. Sample modules for reading JSON, XML and CSV files exist in CoTeTo. This allows flexible processes during development and testing. There is no definition for the structure of the returned data items, since different data sources contain different types of data (tree, table, graph, map). It is the job of the used output *Generator* to understand the data delivered by the used *Data API*.

The most important *Data API* in the Annex 60 context is the interface to the C++ library libSimModel, which handles the SimModel parsing and the mapping to Modelica. Seen from the Python framework and from CoTeTo it defines the data source used to fill the placeholders in the output templates.

Output - Generators Once all relevant data has been loaded into CoTeTo, it is passed to the output component, called *Generator*. We designed the *Generator* to contain all items needed to generate the code for a specific Modelica library. This includes

- filter functions,
- the meta model structure,
- text templates,
- additional configuration and information and
- additional files.

The filter functions, meta model structure and text templates are used and applied by CoTeTo. Additional files like the mapping rules XML file can be stored inside the *Generator*.

We experienced that some data need manipulation that may not fit well into the mapping rule mechanism. For this purpose, *Generators* can include filter functions (Python code) that we call between the data API and the templates. The filters are custom-built to the used library. In our case, they may include simplification of geometric relationships and calculation of

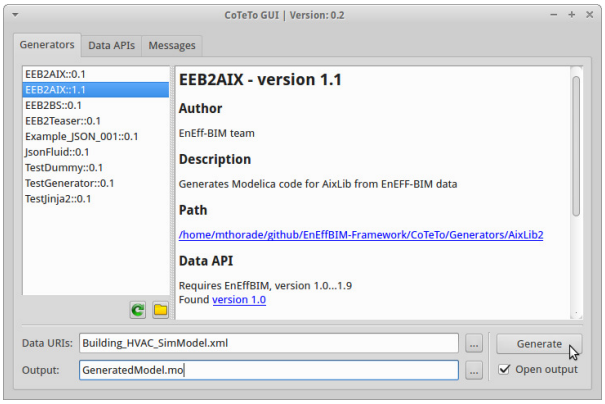

Figure 4. Standalone GUI of CoTeTo

model specific parameters. Another application of filters would be the creation of annotations for the graphical appearance and placement of model components in the Modelica code.

One major challenge in the automated generation of Modelica models is the flexibility of Modelica. Generally said, setting up useful models needs the knowledge of an experienced user. We are following the approach to encapsulate this knowledge in library specific meta-models and templates. One essential task is the appropriate connection of components. The API returns the connection information corresponding to the SimModel ontology, which differs from the one in a Modelica library. The meta model checks if the connection is applicable, if not, it manipulates it.

The text templates are the last step in the process chain. The template engine combines the data structures returned by the *Data API* and possibly manipulated by filters with the text templates to files with valid Modelica code. The templates in a *Generator* can be splitted into several files to ease maintenance.

Generators can be easily exchanged between different installations, as they are simple folders or even zip-files with a defined structure. *Generators* can be maintained and edited with standard system tools like a file manager and a text editor. Creating a new *Generator* is as simple as copying a folder with an existing *Generator* and changing the name or version number in a text file.

Interface and Handling There are currently three ways to use CoTeTo:

- CoTeTo can be imported in Python software as a module library. CoTeTo works both with Python 2.7 and 3.3+. All functions are usable via the modules API.
- A command line interface can be used interactively or called from other software. It allows listing the available *Data APIs* and *Generators*

and executing a *Generator* with a data source URI to produce the text output.

- The graphical user interface (GUI) is implemented using PyQt4. It allows flexible browsing and editing of all components and included files and the execution of selected *Generators*. The GUI can be used as a standalone tool (see Figure 4) or embedded in PyQt4-based applications as a widget.

4 Use Cases

To prove the concept of the toolchain, we have developed several use cases. These use cases are kept simple to ensure the focus on the process. Each use case consists of one single room, according to the description of the validation example of German Guideline VDI 6007 (German Association of Engineers, 2012) and varying HVAC setups. We divide the use cases in two groups, water- and air-based systems. The water-based systems are only meant for heating, while the air based systems also cool and ventilate the room.

The first group of use cases has some basic elements in common. These include a pump, pipes, a PID controlled valve and an expansion vessel. The efficiency of the pump is given depending on the volume flow. The control strategy of the pump includes a night set back, where the volume flow is reduced during nights. We designed the use cases to be controlled by a PID controlled valve. The control variable is the room temperature. Besides these common components the water-based systems differ in the heat generation and heat distribution. The different combinations are as follows:

1.1 Boiler & Radiator
1.2 Boiler & Radiator & Domestic Hot Water System
2.1 Heat Pump & Radiator
2.2 Heat Pump & Floor heating
3.1 CHP & Boiler & Radiator

The second group of use cases consist of two air-based setups. Similar to the first group they have most of the components in common, like air ducts, fans, filter, damper and silencer to account for additional pressure losses. They differ in the purpose of the ventilation system. The first of the two use cases is primarily heating and includes an electrical heater, the other use case is primarily cooling and includes an evaporative, adiabatic cooling device.

The following section presents the first use case (use case 1.1) and the corresponding Modelica code generation using `AixLib` and `BuildingSystems` library. As we focus on the HVAC system, we will describe the code generation for this part of the model only. The thermal zone is currently modeled using the low order model from `AixLib` for both implementations. The hydraulic schema is shown in Figure 5. A gas boiler

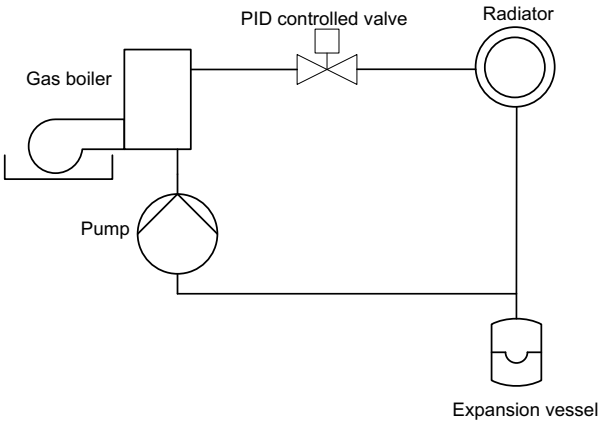

Figure 5. Hydraulic schema of the applied use case

heats the water to a fixed set point temperature. A pump circulates water in the hydraulic system. A radiator emits the heat to the thermal zone. The parametrization of the radiator follows the DIN-EN 442 (DIN German Institute for Standardization, 2015). We designed the radiator to be controlled by a PID controlled valve. The use case is completed by connecting pipes and an expansion vessel.

By calling the API, we load the contained data from the SimXML file into our Python framework. This data is already mapped to the corresponding library as previously described, in our case to `AixLib` or `BuildingSystems`. In addition to the mapped data, the API returns the topology of the use case as a sorted list. The next step is to check if the SimModel topology fits with the Modelica topology in terms of positioning of the components according to the hydraulic schema and correct Modelica connections. By analyzing the Modelica models we identify four different connection types we have to handle in the use case, all types are included in the Modelica Standard Library (MSL). These four connections types correspond to the SimModel connections. The connections (MSL) and their relations to SimModel are as follows:

Table 1. Comparison of Modelica and SimModel connectors

MSL	SimModel
Fluid connector	Water connector (cold, hot)
Thermal connector	Air connector
Real connector	Control connector
Boolean connector	Control connector

The framework connects one component after another according to the given topology, ensuring that the connectors of the models match. In our first use case we have a simple loop and all components, except the expansion vessel, extend from a simple two port model. This is a straight forward approach, as the topology between the hydraulic schema, SimModel and

DOI 10.3384/ecp15118383

Modelica does not differ much. Components that are not considered in SimModel, like the expansion vessel are automatically implemented in every hydraulic loop modeled in Modelica. The meta model contains information about the connection of each model and information about components that need to be inserted automatically. The model for a thermal zone from `AixLib` has two `Thermal` connectors for the implementation of convective and radiative heat sources. Both radiators from the two libraries also have a convective and a radiative thermal connector, while SimModel uses a single air connector. Further, the API passes the information that the radiator and the thermal zone are coupled. The meta model collects and compares all this information and produces a connection between radiator and thermal zone.

More challenging is the correct choice of control systems and their connection to the components. The chosen controller set up in Modelica depends heavily on the used component model. For example, the pump model in `AixLib` has a Boolean input that can turn on and off a night mode with a reduced volume flow. This allows a direct use of `Modelica.Blocks.Sources.BooleanPulse` as a control element. The pump of `BuildingSystems` requires the pressure rise over the pump as a `Real` input. This example shows possible differences in Modelica's control implementations. Typical control strategies, like a night set back, are embedded as templates in the meta model. These strategies are directly mapped to the ones in SimModel.

Once all data is processed, the result is a valid Modelica model. A Modelica representation of this specific use case is given in Figure 6, here using the `BuildingSystems` library. The mentioned pump control is highlighted in red. At this stage of the project the graphical layout of components in Dymola is not supported and needs manual input.

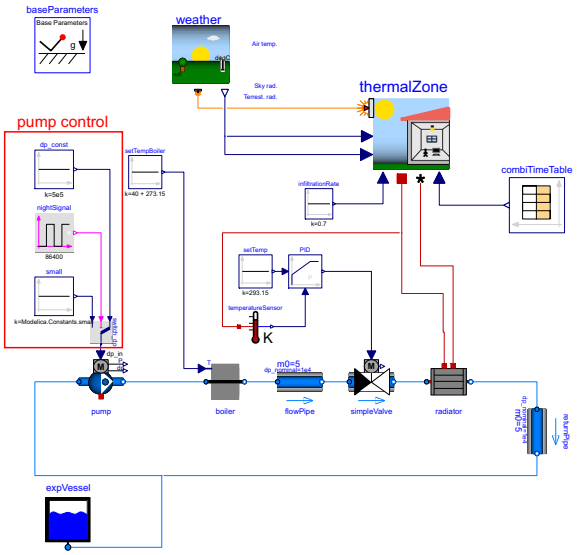

Figure 6. One of several use cases (here shown with BuildingSystems).

5 Summary

5.1 Limitations

As Annex 60 is an ongoing project and all tools are currently under development, we are aware of limitations in the discussed framework. Some of the limitations will be tackled in ongoing work, others are out of the projects scope. The following section provides an overview of known limitations. As the Building Information Model comes in the form of an IFC file, we assume a valid, well formed model. The IFC file is the foundation of the presented toolchain. For example the IFC file has to contain SpaceBoundary entities. Yet, the process is semi-automated and still needs input from the user. Whenever the Modelica libraries change, the mapping rules and in some cases also the templates in CoTeTo have to be adapted. For this reason we enable the use of different versions of Modelica library and the corresponding mapping rules and templates. Although the latest versions of the libraries are used, the initialization of the Modelica model may not provide good starting values. The model still needs fine tuning and the correct choice of initialization values. The generated Modelica model can be seen as a first starting point. This is also true for the graphical arrangement in the used simulation environment, in our case Dymola. For simple models like the Use Case, the representation is fairly straight forward. If we look at more complex systems a meaningful arrangement is more challenging and currently not supported. Further limitations are that currently only two libraries of the Annex 60 project are supported and to-date the full toolchain is only available for a single Use Case. Future work includes the testing of the developed tools with the other Use Cases, as well as setting up more complex and realistic Use Cases.

5.2 Conclusion

This paper presents Modelica code generation for Building Performance Simulation based on Building Information Models. The focus is on an open-source Python framework to connect BIM with Modelica. The work is embedded in IEA-EBC Project "Computational Tools for Building and Community Energy Systems", also known as Annex 60. As Annex 60 is an international project with many participants, our approach is an open, adaptable and integrated toolchain with several standalone usable tools. We developed the toolchain to

be generic and applicable for arbitrary Modelica BPS libraries. In this paper we present a Python framework that basically covers three tasks in the overall toolchain: Process control, information pre-processing and Modelica code generation itself. The framework offers the possibility to access the BIM and control the process from a GUI, command line interface or with use of Python scripts. Covered process control includes the choice of a specific file and applying mapping rules or pre-processing steps for different libraries. The pre-processing includes calculation of library and model specific parameters and creation of graphical annotations. To print out the desired Modelica code, we use a template approach. Templates are easy to use and manipulate for each users needs, without necessarily diving into the code itself. We tested the whole toolchain using a first simple Use Case. Future work will include testing the developed process on more detailed Use Cases. The intent of the project is that all developed tools will be available under an open-source license.

Acknowledgement

This work emerged from the Annex 60 project, an international project conducted under the umbrella of the International Energy Agency (IEA) within the Energy in Buildings and Communities (EBC) Programme. Annex 60 will develop and demonstrate new generation computational tools for building and community energy systems based on Modelica, Functional Mockup Interface and BIM standards.

We gratefully acknowledge financial support by BMWi (German Federal Ministry of Economic Affairs and Energy), promotional references 03ET1177A and 03ET1177D.

References

Ruben Baetens, Roel De Coninck, Filip Jorissen, Damien Picard, Lieve Helsen, and Dirk Saelens. OpenIDEAS - an open framework for integrated district energy assessments. In *Proceedings of the 14th IBPSA Conference*, 2015. (submitted).

Michael Bayer. Mako Templates for Python. `http://www.makotemplates.org/`, 2014. Accessed: 2015-05-13.

Vladimir Bazjanac, Tobias Maile, James O'Donnell, Cody Rose, and Natasa Mrazovic. Data enviroments and processing in semi-automated simulation with EnergyPlus. In *CIB W078-W102: 28th International Conference. CIB, Sophia Antipolis, France*, 2011.

Jun Cao, Tobias Maile, James O'Donnel, Reinhard Wimmer, and Christoph van Treeck. Model transformation from SimModel to Modelica for building energy performance simulation. In *Proceedings of the 5th German-Austrian IBPSA Conference*, pages 242–249, 2014.

Jun Cao, Reinhard Wimmer, Matthis Thorade, Tobias Maile, James O'Donnel, Jörg Rädler, Jérôme Frisch, and Christoph van Treeck. A flexible model transformation to link BIM with different Modelica libraries for building energy performance simulation. In *Proceedings of the 14th IBPSA Conference*, 2015. (submitted).

DIN German Institute for Standardization. Radiators and convectors - part 1: Technical specifications and requirements, 2015. 442 - 1.

Charles Eastman, Paul Teicholz, Rafael Sacks, and Kathleen Liston. *BIM handbook : a guide to building information modeling for owners, managers, designers, engineers and contractors*. Wiley, Hoboken, NJ, 2008. doi:10.1002/9780470261309.

Energy in Buildings and Communities Programme (EBC). IEA EBC Homepage. `http://iea-ebc.org/`. Accessed: 2015-05-13.

Marcus Fuchs, Ana Constantin, Moritz Lauster, Peter Remmen, Rita Streblow, and Dirk Müller. Structuring the building performance Modelica model library AixLib for open collaborative development. In *Proceedings of the 14th IBPSA Conference*, 2015. (submitted).

German Association of Engineers. Calculation of transient thermal response of rooms and buildings - modelling of rooms: VDI 6007-1, 2012. 91.120.10, 91.140.10, 6007-1.

International Organization for Standardization. Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries, 2013. ISO 16739:2013.

Christoph Nytsch-Geusen, Jörg Huber, Manuel Ljubijankic, and Jörg Rädler. Modelica BuildingSystems – eine Modellbibliothek zur Simulation komplexer energietechnischer Gebäudesysteme. *Bauphysik*, 35(1):21–29, 2013. doi:10.1002/bapi.201310045.

James O'Donnell, Richard See, Cody Rose, Tobias Maile, Vladimir Bazjanac, and Philip Haves. SimModel: A domain data model for whole building energy simulation. In *Proceedings of the 12th IBPSA Conference*, pages 382–389, 2011. URL `http://eetd.lbl.gov/node/51892`.

Qt. Qt Cross-platform application and UI development framework. `http://www.qt.io/`, 2015. Accessed: 2015-05-13.

Peter Remmen, Jun Cao, Sebastian Ebertshäuser, Jérôme Frisch, Moritz Lauster, Tobias Maile, James O'Donnell, Sergio Pinheiro, Jörg Rädler, Rita Streblow, Matthis Thorade, Reinhard Wimmer, Dirk Müller, Christoph Nytsch-Geusen, and Christoph van Treeck. An open framework for integrated BIM-based building performance simulation using Modelica. In *Proceedings of the 14th IBPSA Conference*, 2015. (submitted).

Armin Ronacher. Jinja2 Templates for Python. `http://jinja.pocoo.org/`, 2014. Accessed: 2015-05-13.

Cody M. Rose and Vladimir Bazjanac. An algorithm to generate space boundaries for building energy simulation. *Engineering with Computers*, 31(2):271–280, 2015. doi:10.1007/s00366-013-0347-5.

Michael Wetter and Christoph van Treeck. IEA Annex 60. http://www.iea-annex60.org/, 2012. Accessed: 2015-05-13.

Michael Wetter, Wangda Zuo, Thierry Stephane Nouidui, and Xiufeng Pang. Modelica Buildings library. *Journal of Building Performance Simulation*, 7(4):253–270, 2014. doi:10.1080/19401493.2013.765506.

Reinhard Wimmer, Tobias Maile, James O'Donnell, Jun Cao, and Christoph van Treeck. Data-requirements specification to support BIM-based HVAC-definitions in Modelica. In *Proceedings of the 5th German-Austrian IBPSA Conference*, pages 99–107, 2014.

Reinhard Wimmer, Jun Cao, Peter Remmen, Tobias Maile, James O'Donnel, Jérôme Frisch, Rita Streblow, Dirk Müller, and Christoph van Treeck. Implementation of advanced BIM-based mapping rules for automated conversion to Modelica. In *Proceedings of the 14th IBPSA Conference*, 2015. (submitted).