Abstract
Building energy modeling (BEM) has been widely used by researchers, regulators, and engineers to quantify building energy performance. Quality assurance (QA) and quality control (QC) of model performance are essential parts of such an analysis. Currently, QA/QC is done in a manual and ad hoc manner, which is tedious, error-prone, and time-consuming when verifying a large number of models. To solve these challenges, we propose a data-driven building performance verification framework (ANIMATE), which can be used as an add-on tool to automatically verify building energy model results (especially time-series output-based verification of control requirements). While this framework was designed for verifying energy model performance, it can be extended for other applications such as BEM software testing and performance verification of real buildings in the field.

Key Innovations
- Designed a data-driven performance verification framework for BEM
- Can be used as an add-on tool to automatically QA/QC building energy model results (especially time-series data)

Practical Implications
Performance verification using time-series data is always crucial in BEM practices, especially for control requirements. Modelers and practitioners can potentially use this framework to automatically verify building energy model performance, especially for specific control requirements.

1. Introduction
Buildings account for 39% of the energy usage in the U.S. and more than 20% globally (U.S. EIA 2019a,b). Building energy modeling (BEM) has been widely used to quantify the energy impact of new building technologies, energy policy development, and other virtual demonstrations of building energy usage. Quality assurance (QA) and quality control (QC) are essential parts of such BEM analysis. This is especially true for whole-building energy modeling, which mostly deals with complex models, driven by a very large number of input data. QA in BEM is defined in Karpman and Rosenberg (2020) as “activities that confirm that the BEM program’s technical requirements result in the expected performance outcomes and setting the minimum qualifications for professionals who develop energy models and perform project reviews,” while Tian et al. (2009) mention that “the American Society for Quality defines ‘quality control’ as ‘the observation techniques and activities used to fulfill requirements for quality’ (ASQ 2008). Quality control can reduce scatter and uncertainty in simulation results.”

Kim et al. (2019) define building faults as “improper or undesirable operations of building systems and equipment.” As such, a parallel can be drawn between building faults and simulation errors related to improper or undesirable operation of building systems or equipment. Throughout the years, QA/QC approaches to BEM have been refined and have taken different forms: input-based, output-based, and even comparative output-based verifications (i.e., benchmarking). Input-based approaches tend to provide feedback based on the overall set or a subset of inputs used by the energy modeler, while output-based approaches provide feedback based on simulation results, whether it is based on aggregated energy use results or performance-oriented metrics such as load hours not met by the heating, ventilation, and air conditioning systems. Often, the two approaches are combined.

Following Frank et al. (2019), output-based verifications would fall under the “outcome-based” fault category. Input- and output-based verification cannot identify behavior-based faults, which are “the presence of improper or undesired behavior during the operation of a system or piece of equipment” (Frank et al. 2019). To do so, time-series outputs from simulation need to be analyzed, an example being the identification of systems providing both unwanted heating and cooling simultaneously. To specify such faults, Lei et al. (2021) developed a formalism to systematically derive customized fault definitions for output-based verification with the limited scope of air handling unit control logic program problems. The authors are not aware of any publicly available framework allowing energy modelers to systematically perform such verifications for a wider range of systems and scopes in buildings.

Another example of an output-based verification process is the one used by the Open Building Control platform (Wetter et al. 2020). The process compares trended time-series output to time-series from a reference case to ensure that control sequences are defined correctly. While this platform is aimed toward building control, one could apply such a QA/QC approach to whole-building simulation. However, even though this approach compares time-series, it still cannot identify behavior-based faults, and it requires a fault-free reference case to be manually verified in advance.
Vega et al. (2014) describe comparative output-based verification approaches used by practitioners where specific metrics are computed for each simulation. If values differ significantly from ones obtained for similar simulations, the simulation is flagged for additional review. In a sense, this corresponds to a benchmarking approach to identify outliers. It contrasts input- and output-based verifications that are only performed on individual models.

Balbach and Bosworth (2016) provide a detailed overview of the most popular QA/QC input-based verifications (e.g., eQUEST QC reporting tool, simulation checklists (BEMLibrary 2020)) and output-based verifications tools (e.g., Xcel EDA’s OpenStudio reporting measure) applied to building energy simulation publicly available at the time. Despite some effort to standardize simulation output reporting through NREL 2016 and COMNET 2012, which would provide a consistent way of performing input- and output-based verifications, the authors point out that most tools are insufficient, and that because of the lack of a standardized approach, most energy modelers tend to develop ad hoc spreadsheet tools to meet their needs. Balbach and Bosworth (2016) go on to outline some key characteristics for a robust and generalist QA/QC framework for BEM: transparency, repeatability/reproducibility, and automation.

In summary, the QA/QC approaches identified in this review rely heavily on simulation inputs and/or reported (summarized) outputs. While this seems like adequate evidence at first glance, it assumes that 1) the simulation software or reference cases are free of errors, which is not always the case, and 2) there is always a unique (or set of unique) and consistent approach(es) that can be used to gauge whether simulation results are correct just by reviewing inputs. In most instances, QA/QC approaches require a modeler or reviewer to spend considerable time reviewing inputs, outputs, and/or metrics to ensure that the outcome of a simulation is acceptable. Unfortunately, both 1) and 2) are rarely true because 1) simulation engines are constantly evolving (bug fixes and new features), so the reference cases might be not free of errors, and 2) time-series output verification (with a combination of summarized inputs and outputs) is the only way to ensure that models behave as expected. Reviewing only model inputs will not guarantee model performance.

To address these challenges/gaps, we propose a dA-ta-driveN building perforMance verification framEwork (ANIMATE) that enables automatic control performance verification. To further introduce this framework, we organized this paper as follows: Section 2 elaborates on the framework design and key steps in the workflow; Section 3 explains the key components of the framework, which contains the knowledge/algorithms used to verify specific performance requirements; Section 4 further explains the framework implementation details; Section 5 presents a few examples to demonstrate how the framework could be used; and Section 6 summarizes the framework and elaborates on its potential uses and future development.

2. Methods

The concept design of ANIMATE is based on the needs of 1) having verification performed through time-series data, and 2) automating such performance verification. The initial motivation for this work was the verification of performance of BEM models against certain control requirements.

As shown in Figure 1, such a process proceeds as follows: building energy model creation, simulation and output dataset generation, data analysis, and finally result reporting. Typically, the last three steps are performed manually. The proposed framework handles all four steps at once, automated as one integrated process.

We designed this framework around three key features: 1) Knowledge integration: The framework aims to facilitate the reuse of knowledge/expertise that is used for verifying building control performance, and to collect that knowledge in a centralized location. In this way, the knowledge base can be easily maintained and extended. This helps knowledge transfer as well as exchange. 2) Analytics: The framework formalizes the process of performance analysis and assessment and ensures the quality of verification. It uses the simulation results for the verification process; this approach is conceptually similar to the commissioning process in the field. The trend data collected through submetering a model or a real building is key for the actual performance assessment. 3) Automated process: The framework generalizes the traditional manual verification process into a computer-executable procedure. A manual process is always challenging for QC. The automation not only saves time for verification and reduces human error, but also promotes the generalization of expertise used for performance verification and the transfer of knowledge from one specific model or building to broader applications.

![Figure 1 Conceptual design of the framework and its interaction for performance verification](https://doi.org/10.26868/25222708.2021.30725)
verification tasks. For example, it could be used by software developers to conduct comprehensive unit testing to verify the performance of BEM software after releasing a new version; it could be used by building modelers to verify if a certain control implementation performs as expected; it could be used by building operators to analyze trend data from the building automation system to verify if certain control requirements are properly implemented.

With the concept design, the framework is further detailed with three major components:

- **Control performance verification algorithm library** (or verification library) – contains algorithms (such as empirical rules or artificial intelligence models) to verify control requirements
- **Automated model preparation process / test case generation** (an automated process to prepare BEM model for reporting selective time-series results)
- **Standardized performance evaluation and reporting process**

Figure 2 illustrates those components, their key elements, and their interactions. Each component consists of several elements, and these elements seamlessly interact with other elements in the same or another component block to process the information and execute the performance verification. An example is given below to further explain the detailed steps of logic flow (for verifying the performance of supply air temperature reset control in a simulation model).

1. Develop verification algorithm: The verification algorithm (e.g., supply air temperature reset control verification) needs to be developed and implemented via one of the three verification types (see the three verification types in Section 3).
2. Gather input data: Prerequisites of the verification process include a simulation model (e.g., a large office building created in EnergyPlus) that includes a specific (or list of) control requirement(s) (e.g., supply air temperature reset control).
3. Instantiate verification items: When, for example, the process of verifying the implementation of supply air temperature (SAT) reset control starts, the first step is to instantiate model-specific verification items. That is, take the user’s requirements (SAT reset is to be verified) and the simulation model as inputs, query the library and retrieve the information about the simulation i/o data requirements for the SAT reset control verification algorithm, then reason through the simulation model and identify and specify all application instances (referred to as verification items hereafter) of this algorithm and the corresponding data points variables with a JavaScript Object Notation (JSON) schema.
4. Revise building energy model: In this step, verification items are passed into the revised energy models to enable it to generate the required output to perform the item verification.
5. Generate simulation dataset: This step runs the instrumented model simulation to generate the simulation outputs file.
6. Conduct performance verification: In this step, the simulation outputs file is read based on requirements of the verification items and then the framework interacts with the library to execute the implemented verification algorithm of the SAT reset control. The algorithm will determine whether the model performance meets the control requirement and/or performs as expected. The process obtains and outputs the verification results, and generates a brief report with illustrative time-series plots.

3. The Verification Library

While the goal of the framework is to perform building model performance verification in a generic manner, the verification library has been developed initially to target the correct implementation of control requirements. For example, there are significant numbers of control-related requirements in building energy codes and standards (e.g., ASHRAE Standard 90.1 and the International Energy Conservation Code).

To create the verification library, we first reviewed all control-related code requirements and tabulated them. Then, they were sorted to identify the most impactful ones in terms of their overall impact on building energy use. A few requirements were selected for implementation, referred to as “verification items” herein. The review process helped identify that multiple approaches would have to be developed to carry out performance-based verifications. These approaches are described below:

1. **Logical expressions**: Verification items (e.g., for verifying the correct functioning of a specific air-side economizer operational scheme) can be achieved through the evaluation of logical expressions. In such an instance (as shown in Figure 3), a code requirement or a specific sequence of operation (SoO) is first translated into a pseudo-code, which is then implemented in the library as a single, or multiple, logical expression(s), which is evaluated for each time-series data point and compared with an expected outcome.

2. **Procedural empirical knowledge**: Some verifications items require more complex means of verification. A good example of such an item is daylight responsive electric light dimming controls. These items can be verified by using procedural empirical knowledge or by performing machine-learning-based verifications. The former uses knowledge gathered a priori in the verification procedure to see if the data complies with a particular requirement or control strategy. For example, in the case of daylighting dimming control strategies, we know a priori that lighting system output should correlate with the amount of daylighting detected by a photosensor.

3. **Machine-learning-based verification**: This can be used in place of procedural empirical knowledge verifications when 1) logical expressions cannot be used to accurately perform verification of a specific item; 2) when knowledge for a specific item is difficult to formulate and when relatively large amount of labeled data exists so a machine learning model can be trained to identify if a particular dataset complies with a specific code requirement. Section 5 provides an example of the usage of a convolutional neural network to identify whether a model uses one of two lighting control strategies. This type of strategy is highly dependent on the quality of the labeled data used to develop the model and tends to be more resource-intensive than other types of verifications.

<table>
<thead>
<tr>
<th>Brief Description</th>
<th>Related Code Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economizer Integration: Non-Integrated</td>
<td>Typical requirement as follow: &quot;Economizer systems shall be integrated with the mechanical cooling system and be capable of providing partial cooling even when additional mechanical cooling is required to meet the remainder of the cooling load.&quot; Section 6.5.1.3 in 90-2004</td>
</tr>
<tr>
<td>Economizer Operation</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 3 – An example of using logical expression to interpret energy code requirement to verification code](image)

4. Framework implementation

The implementation of the framework consists of the development of an information schema and a software with an extensible interface. The developed information schema is used to instantiate specifications of verification cases. For each item to be verified, one instance of this information schema is created and presented to the automated verification framework as an input to specify the following categories of information:

- Categorization and narrative description of the instruction item to be verified.
- What information needs to be instrumented into the simulation model in order to obtain the desired output for verification.
- What simulation output data needs to be extracted to conduct the performance verification.
• How to perform verification in terms of which class in the implemented verification library to call.

This schema is implemented as a JSON schema and can be used to automatically validate the JSON file for the instantiated verification items. For example, an excerpt of the instantiated schema for verifying continuous dimming control requirement is shown in Figure 4.

The software is developed using the Python programming language and has three main features: 1) it implements the proposed automated verification workflow as shown in Figure 2; 2) it hosts the implementation of verification library, and 3) it specifies the programming interface to extend the verification library with implementations of new procedural empirical knowledge-based and machine-learning-based verification items.

Different steps (activities) of the verification workflow are implemented into modularized and object-oriented classes and subclasses. Modularity and object-orientation makes it easy to refine and extend the framework with improved and new programming features in the workflow and can be used to automatically validate the JSON file for the instantiated verification items. For example, an excerpt of the instantiated schema for verifying continuous dimming control requirement is shown in Figure 4.

The software is developed using the Python programming language and has three main features: 1) it implements the proposed automated verification workflow as shown in Figure 2; 2) it hosts the implementation of verification library, and 3) it specifies the programming interface to extend the verification library with implementations of new procedural empirical knowledge-based and machine-learning-based verification items.

Different steps (activities) of the verification workflow are implemented into modularized and object-oriented classes and subclasses. Modularity and object-orientation makes it easy to refine and extend the framework with improved and new programming features in the workflow (Lei 2019). A central workflow controller class is implemented to assemble different parts of the framework and execute them properly.

```
class CheckLibBase(ABC):
    """Abstract class defining interfaces for item-specific verification classes"""
    points = None

    def __init__(self, df: pd.DataFrame, params=None):
        full_df = df.copy(deep=True)
        if params is not None:
            for k, v in params.items():
                full_df[k] = v
        col_list = full_df.columns.values.tolist()
        if not set(self.points_list).issubset(set(col_list)):
            print("Dataset is not sufficient for running (self.__class__",
                  "name__)")
            print(set(col_list))
            self.df = full_df[set(self.points_list]
            self.verify()

    @property
    def points_list(self) -> List[str]:
        """returns required data points list for the verification"
        return self.points

    @property
    def get_checks(self):
        """returns Boolean and detailed verification outcome"
        return self.check_bool(), self.check_detail()

    @abstractmethod
    def check_bool(self) -> bool:
        """returns Boolean verification outcome"
        pass

    @abstractmethod
    def check_detail(self) -> Dict:
        """returns detailed verification outcome report as a Dict"
        pass

    @abstractmethod
    def verify(self):
        """implementation of the verification algorithm"
        pass
```

Figure 4 – Excerpt of an instantiated verification case specification: 1 contains elements about descriptions and categorization of the verification item; 2 contains elements about verification data points mapping; 3 contains elements about data points description; 4 contains pointer to knowledgebase class that encodes the verification algorithm for this item.

The implementation of the verification library is guarded by an abstract CheckLibBase interface (a simplified representation is shown Figure 5) that specifies the implementation requirements for all instantiateable classes being added to this library, each representing the verification routine of a verification item. In the verification library implementation, implemented verification items are Python classes, all inheriting this CheckLibBase interface. The class names of these subclasses are used as identifiers in the schema instances.

Figure 5 – Excerpt of the CheckLibBase interface: 1 field for verification classes to specify data points requirement; 2 constructor to parse simulation output data and check if required data points are included in the output, and if so, run the verification algorithm; 3 defined properties interface; 4 abstract methods interface to be implemented by verification classes.
to specify which implemented verification item is going to be used to verify that specific case.

To extend the verification library with new verification routines, one only needs to add a new Python class extending the CheckLibBase interface and implement the methods required by this interface.

5. Demonstration Examples

The following examples demonstrate how ANIMATE can be used to verify performance for specific control requirements. Table 1 presents two examples of verification that were implemented as part of the proof of concept of the framework.

<table>
<thead>
<tr>
<th>ID</th>
<th>Verification Item</th>
<th>Description</th>
<th>Building Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Continuous dimming</td>
<td>Check that lighting systems are controlled using a continuous dimming strategy</td>
<td>Medium office</td>
</tr>
<tr>
<td>2</td>
<td>Non-integrated economizer control</td>
<td>Check that economizer operation appropriately reflects the operation of a non-integrated economizer</td>
<td>Primary school</td>
</tr>
</tbody>
</table>

The tests were conducted using the U.S. Department of Energy’s Commercial Prototype Building Models, developed and maintained by Pacific Northwest National Laboratory (Athalye et al. 2017; Thornton et al. 2011). The medium office model (see Figure 6) was selected for evaluating the lighting/daylighting controls, and the primary school model was selected for evaluating non-integrated economizer control. Table 1 provides a short description of the verification being performed as well as the general strategy used to do so. The examples include both a simple rule-based verification as well as more advanced verifications using applied machine learning techniques. The authors want to point out that the examples are directly derived from building energy codes requirements.

Figure 6 – Axonometric rendering of the medium office model

Non-integrated Economizer

Economizer integration with mechanical cooling equipment means that an air-side economizer is able to operate in conjunction with the system’s mechanical cooling, while a non-integrated economizer will only operate if mechanical cooling is not operating. Recent building energy codes require economizers to be integrated (ASHRAE 2019). However, older codes don’t always require it. When comparing savings from one building energy code version to another, it is important that the baseline condition be modeled appropriately. This test intends to verify that air-side economizers are modeled as non-integrated with the mechanical cooling.

Based on the predefined pseudo-code shown in Figure 3, a logical expression-based verification algorithm was developed for ANIMATE and stored in the verification library. This can be considered as a generic pre-developed verification algorithm for verifying economizer operation. During the verification process, the framework first identifies the control requirement (i.e., to verify the non-economizer operation) requested by users for verification. Then, based on this request, it calls out this verification algorithm (i.e., the logical expression for non-economizer operation verification) to evaluate the result/data from each simulation timestep. Specifically, the verification algorithm looks at the system’s cooling coil output and its outdoor air flow rate to assess whether a non-integrated economizer is indeed being simulated or not. After the data is evaluated, the framework lets the user know whether the requirement that is being investigated is met for the whole simulation period. Additionally, the framework can generate graphical representations of the variables used for the verification and of the verification or assertion result. Figure 7-a and -c show that the system’s mechanical cooling is operating (“Cool_sys_out” in the top chart) when the outside air flow rate is greater than the system minimum, which indicates that the economizer is operating as an integrated economizer. Figure 7-b shows that for this example the assertion being evaluated is false (equal to 0) between 9 am and 8 pm. This means that during all simulation timesteps within this period, the economizer is operating as an integrated economizer. Note that this figure only shows a sample of the whole verification period (whole year).

Continuous Dimming

Recent building energy codes require lighting systems to be controlled to reduce their lighting power based on the amount of daylight sensed by photocells in a continuous and linear fashion, all the way down to a minimum fraction (ASHRAE 2019). This strategy differs from other common control strategies such as stepped control, where specific electric lighting levels are output at different thresholds of daylight sensed by photocells. A continuous dimming strategy can typically provide additional energy savings over stepped controlled strategies, so whether it is for building energy code compliance or energy code benchmarking, it is important to make sure that savings are appropriately accounted for.

A convolutional neural network (CNN) was trained and tested on normalized lighting profiles from actual simulations, the goal being to automatically identify cases using continuous dimming strategies. The result shows that more than 95% of cases are correctly categorized. For example, the profile in Figure 8 was labeled by the CNN as continuous dimming, while the profile Figure 9 was labeled as stepped dimming.
Figure 7 (a,b,c) – Plots from non-integrated economizer verification: (a) cooling coil energy time series; (b) trigger flag time series of the verification rule; (c) outdoor air flow rate time series with minimum outdoor air value reference

Figure 8 – Normalized lighting profile (normalized power) for a continuous control strategy

Figure 9 – Normalized lighting profile (normalized power) for a stepped control strategy

6. Conclusion

In this paper, we have presented a data-driven performance verification framework that enables automatic control performance verification. This framework comprises three major components: 1) Verification (algorithm) Library, 2) an automated model preparation process/test case generation, and 3) a standardized performance evaluation and reporting process. Two verification examples have been used to demonstrate the usage of this framework. Based on that, we envision this framework can be potentially used for the following application:

1. Automate the BEM performance verification process, enabling building energy modelers to quickly verify their energy model performance via a systematic checking process.

2. Improve BEM engine accuracy through automatic QA/QC and create a systematic way of verifying BEM performance for software developers whenever there is a version upgrade. This verification will ensure that the performance check process is automated and maintains the BEM software’s consistency and accuracy.

3. Provide a living verification library that collects and accumulates knowledge and algorithms developed and used for performance verification.

This work was initially funded to investigate the possibility of QA/QC of many building energy models automatically in 2020. This paper documented the
preliminary outcome of the framework development. It allowed us to assess its potential and realize that it can have a strong impact on both energy modeling and actual building performance verification (beyond the scope of large-scale modeling and prototype model performance verification). We are further developing this framework, and more details will be updated via the following link on GitHub https://github.com/pnnl/ANIMATE.

Therefore, we have further expanded the framework's objective to be a more generic tool for building performance verification and to cover individual or a collection of models and/or actual buildings. The outcome of the ongoing work will be reported in future publications.

7. Acknowledgement

This work was supported by the Building Energy Codes Program (BECP) and the Building Energy Modeling (BEM) sub-program of the Emerging Technologies Program, at Building Technologies Office, of the U.S. Department of Energy under Contract DE-AC05-76RL01830.

References


