Scalable smart building simulation

Emiel Pasman

August 22, 2019

Supervisors: M. Medema, Prof. Dr. A. Lazovik

Contents

1 Introduction

A smart building is a building in which operations are automatically controlled. Usually, sensors are used to make intelligent decisions about these operations, in order to ensure comfortable conditions for users with minimal energy consumption. Smart buildings are becoming increasingly important due to the need of saving energy. Buildings consume an estimated 20-40% of electricity and part of this energy is wasted [1]. Smart buildings should eliminate this waste by controlling systems such as lighting, heating and ventilation to provide optimal comfort only when people are present and in the most efficient way possible.

With the information provided by sensors, some control system is to make decisions on the optimal operations of the building. To test control algorithms, a simulation can be used to save time and resources. In order to make this possible, the aim of this project is to create a program that can simulate smart buildings in a scalable manner.

Simulation of smart buildings and city districts is currently a topic of active research. The optimization of energy usage has become increasingly important. Simulating the effectiveness of control policies that aim to reduce it is not only a useful way of determining optimal policies without testing them in a physical environment, it can also be used to predict the need for energy in the future allowing the needs of the users of buildings to be met more precisely and preventing the generation of excess energy. With simulations, the resulting energy usage, user comfort and other parameters could then be used to compare different methods of control.

Simulating larger scale models is difficult due to the dependencies in the parameters to be predicted. For this bachelor's thesis, a program was to be made that enables scalable simulation of smart buildings by exploiting independence in parts of the data using parallelization. In order to this, information is gathered to determine how simulations can be made scalable and how a smart building specifically could be simulated efficiently. Then, an architecture is decided upon based on this information and a Scala implementation made. This implementation will be tested for results. This way, a method should be found and tested to effectively make simulations of smart buildings scalable.

2 Related work

Publications on simulation of smart buildings and cities focus on the accuracy of the sensor values in it, which can be achieved by using a detailed model of the building or by analyzing historical data [2]. The method of acquiring correct sensor states is not a topic of focus of this project. Each sensor will have its own function that computes the correct state based on the state of actuators, which

can consist of some computations or a request to another program to compute a value. These functions must be provided to the program in order to obtain correct results, and tests performed using the program for this project use simple functions that do not represent any physical or statistical relationship.

2.1 Parallelizing simulation

As performance of simulations is the main area of research related to this project, different types of simulation will be explored in order to determine the different possibilities with regard to modelling. In order for applications to be scalable, they must be parallelizable. However, in order for simulations to be accurate they must correctly represent the causal relations between events that occur as time goes on. Research on parallelizing such simulations is done in the field of Parallel Discrete Event Simulation (PDES), and within this field methods can be divided into two categories: optimistic and conservative methods [3].

Conservative methods involve making processes wait at synchronization points if it might be affected other processes [4]. Once it is certain that no potential effects have been left out, the process will continue to the next point. In parallel programs, blocking is undesirable as there may not be enough tasks available for all processors if threads are blocked. However, the more processes are present, the more likely it is that there will still be some task to perform. Therefore, a simulation with many elements can still perform well in the conservative method.

The optimistic approach is to let them continue processing new events and undoing their effect up to when the synchronization should have happened, if necessary [5]. For example, in the case of a smart building, a light sensor might be providing information about the light levels based at each point in time in the simulation. The sensor may already have computed the light level of time t=5 when the control algorithm decides that at time t=2 a light should be turned on. In this case, all events processed by the sensor after t=2 are undone and the event representing the light turning on is processed. After this, the process will continue from that event, having added the previously processed events to the queue again. This approach is most useful in situations in which few rollbacks are necessary.

2.2 Actors

In order to create concurrently running processes that communicate through messages regardless of the physical location of each process and are resilient to failures, the Akka library can be used. Akka is a library that provides an implementation of actors made to allow for the creation of distributed applications. It is available for Scala and Java.

2.3 Rules

In a smart building, users set rules which the system can use to determine what the state of the actuators within the building should be based on the state of the sensors. There are different ways to formulate these rules, convert them into a solvable problem and solve the problem. One way to do so is by using constraint satisfaction problem (CSPs) [6] [7]. A constraint satisfaction problem consists of a number of variables, each with a given domain, and constraints. A solver uses some search strategy to find values for each variable within their domains such that all constraints are satisfied. An objective function can be used to give each solution a value which can then be minimized or maximized. This is done by optimizing a variable for a single objective or by optimizing a scalar combination of multiple variables, where the scalar values represent weights. For example, energy consumption and user comfort could each be given their own weights, resulting in a solution that attempts to balance the two.

2.4 Constraint programming

Some programming languages and libraries allow for simple creation and solution of CSP models. One such library for Java is the Choco solver, which allows users to create a model object, post constraints to it and then solve, with many options for both constraint types and search strategy.

3 Architecture

To model actuators and sensors in the building, events can be simulated across multiple processes representing them, similar to a PDES. All actuators and sensors are processes that communicate through messages. To solve the synchronization problem brought up in the related work section, a conservative approach is used. Whenever a sensor sends a value, it waits for influences from any actuators that could change states as a result. Sensors only send values as a response to a message containing a vector clock of last update times of actuator states. With this information, the sensor can determine if it has the most recent values from each actuator that it is receiving data from. Actuators are, as a result, always synchronized even without checking clocks as actuators only receive their new states as a consequence of a sensor's state change.

3.1 Synchronization

The conservative method of PDES means that a process can only continue when it is certain that no event can be added in the future with a lower timestamp than the one that is currently due to be processed. In the case of this smart building simulation, this is guaranteed if sensors know which actuators to wait for and there is a single process that requests values from all sensors. After the sensors' values are requested, they cannot receive events preceding the time of

the request except from actuators, since requests for sensor values are sent from a single process in chronological order each time the rules are evaluated and sensors never send messages to each other directly.

If sensors all have their own event queues like most processes in a simulation, the process of not sending messages early would be much more difficult, as sensors may still be influenced by other sensor values that are sent first and thus must wait for these. However, it cannot know that all preceding events have been processed without sorting all sensor events to ensure that its turn is now. Therefore, having a single process that maintains a sorted queue of sensor events is preferable. Similarly, if there was no clock to indicate which actuators have been updated at what timestep, sensors could never be certain that it is safe to proceed. At any point in time, the sensor could still be receiving a new actuator value in the future so sending a state would not be safe.

3.2 Independent simulations

With this structure to the processes, all sensors can update their values simultaneously as they should all be receiving updated actuator values shortly after the rules have been applied. However, in order to parallelize further, it is possible to also solve multiple CSPs at the same time if none of the sensors and actuators influence each other's value in any way. If there is an independent part of the model, solving these independent parts can happen in parallel as well. Two processes are considered independent if the state of one process cannot be influenced by the other, directly or indirectly. If a sensor, through the rules, affects an actuator that, in turn, affects a different sensors these sensors must still be considered dependent on each other as the state of the latter sensor can be influenced by the state of the first through a change in the rules. This particular dependency relation is therefore transitive.

The independent parts of the simulation are split by creating different sets of sensors and actuators that are unaware of each other. Splitting up the simulations prevents these processes from having to wait for any that have no dependency relation, increasing the parallel performance. The resulting CSPs to be solved will be smaller, containing only the sensors with values that depend on each other.

The program requires all information on sensors, actuators and rules as input. This means that each sensor must be given its own initial state, a list of influences each actuator connected to it has on the sensor's state as a function of time and that actuator's state and a function that returns the effect of the environment or use case. Additionally, a number should be provided for each actuator that will be used to scale the actuator's state in order to determine the energy cost. With this input, a model can be created of the building and any parameters of interest can be messaged or logged by the processes in order

to create output.

4 Implementation

The simulation program is made in the Scala language. In the program, sensors and actuators are modeled as actors. There are also evaluators that evaluate the rules and ask for sensor values in the right order. In the program, each of these actors are of a class derived from the Akka actor class.

4.1 Sensors

Sensor actors contain and send information on the state of the building. Because the sensor actors are part of a simulation, no physical environment exists which information can be taken from. Therefore, the sensor actors receive all information that is needed to determine the value of the parameter that is to be measured in the form of functions that can be evaluated by the sensor at points in time when the state must be sent. Sensors can have either a state that changes only when events occur or continuously. Each sensor has its own local time. Sensors with a continuously changing state send their state to the evaluator at regular intervals, as opposed to sending state whenever a change occurs. These regular intervals are specified when the objects are created. After the state and the accompanying timestamp have been sent, the sensor must wait until the state changes in actuators brought about by the evaluator have been received, as allowing each sensor to send its state again immediately would allow the sensors to advance in time without knowing the effect of state changes in actuators within that timeframe, breaking causal ordering.

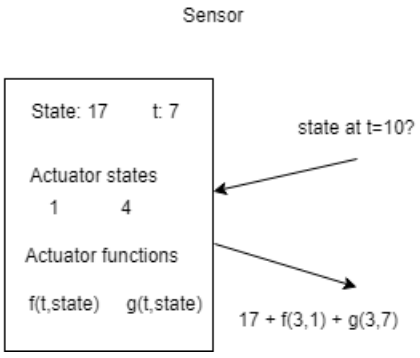

Figure 1: Sensor workings

4.2 Actuators

Actuator actors receive their desired state and send it to any sensors that may need it. These actors hold their state and know the sensors that must receive the state, but do nothing further as the sensors calculate their effect based on the state they receive. With a completely simulated environment, the actuator actors serve little purpose as the information about actuator states could be relayed to the sensors directly. The only difference is that messages can be sent to multiple sensors by multiple actors at the same time. Another advantage of this approach is the resemblance to a real situation which is useful when the program might interact with other software or hardware systems.

4.3 Evaluators

Evaluator actors receive the state of sensors and determines what the state of actuators in the system should be, after which they send new states to these actuators. Multiple evaluators can exist if some part of the rules can be evaluated independently of the rest, as this allows concurrent running of multiple evaluations. This actor contains the most recent values received from each connected sensor and a function that converts these values into desired actuator states. This function first determines which constraints should be active currently. After this, it creates a new model in the Choco solver, in which actuator states are considered variables and constraints are the rules that are currently active.

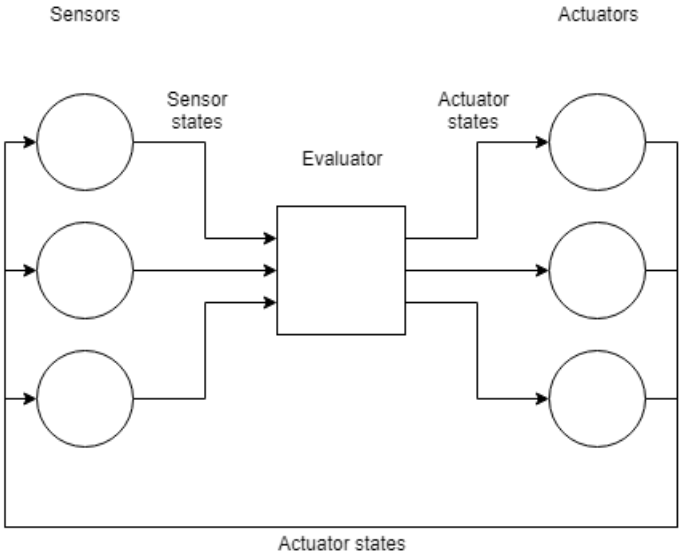

Figure 2: Actors and the states they send

Rules must be evaluated whenever a sensor sends a value and sent sensor values should be evaluated in the order of their timestamps. To guarantee this, evaluation should wait until all effects of actuator state changes have been calculated. To allow for this, the evaluator asks sensors for the values in accordance with the event it is processing, also sending a vector clock with a time value for each of the actuators. The sensor should then reply with its current state after it has waited for its clock to be synchronized again.

Thus, a time step is resolved as follows: first, the evaluator dequeues an event from the queue of events. The event should be processed and as a result, the evaluator requests the values of one or more sensors depending on their update intervals and the current time. These values are then used to replace the old values of these sensors, after which the rules are applied to them, resulting in states for the actuators that should be sent out if they are different than in the previous step. The actuators store these new states and send them to all affected sensors.

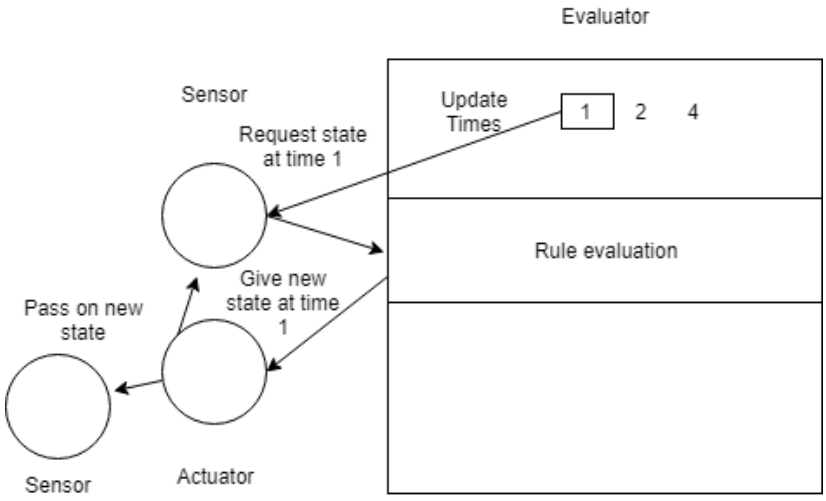

Figure 3: Evaluator processing an event

5 Results

In order to determine the performance of the application, tests will be performed to determine the runtime of the program and the effects on it of changing the number of independent rooms and the number of events. Each test will be performed with different numbers of cores to determine the speedup at different

numbers of cores. All tests were performed on an Intel i5 4460 at 3.2 GHz.

The test all used a virtual 'room' that consists of 4 sensors and actuators. Each sensor calculates its new value based on one of the actuators by scaling it and then adding that number to the old sensor value.

There is a set of rules for each sensor and the connected actuator, specifying 3 ranges in which the value of the sensor can fall and a value for the actuator for each of these sensor value ranges.

5.1 Number of evaluators

To test how the program performs when there are many independent parts of the simulation, this scenario consists of a varying number of rooms with 4 sensors and actuators in each of them. Each of these rooms have their own evaluator, allowing the program to independently solve for each. 10 Events will be processed by each of these evaluators.

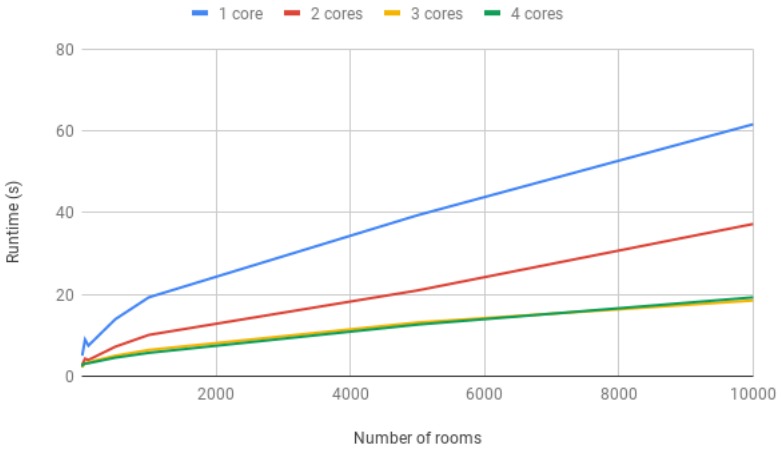

Figure 4: Runtime for different numbers of evaluators

This figure shows that as the number of rooms increases, the runtime increases linearly. However, the rate of change for the higher numbers of rooms is lower than 1, likely due to the parts of the program that have a runtime constant with the number of rooms becoming less and less influential on the total runtime. There is a clear problem with the results of this test: the results for 4 cores are almost the same as for 3 cores. It is unclear why this is the case, and the next test has the same problem. One possible explanation of this could be that other tasks running in the background on the machine are preventing the efficient use of the maximum number of cores.

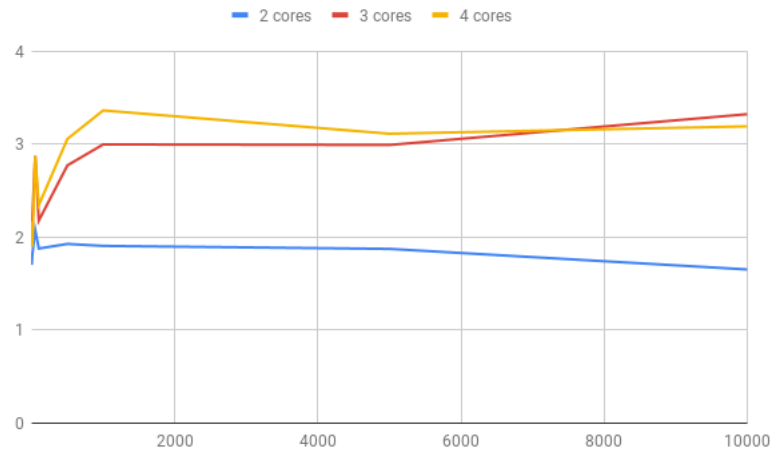

Figure 5: Speedup for different numbers of evaluators

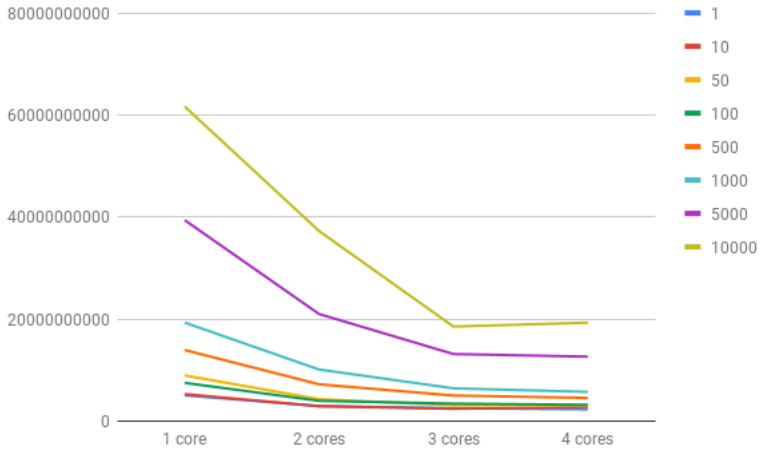

Figure 6: Runtime for different numbers of cores

The speedup at 3 and 4 cores is, as visible from the previous figure, the same. However, the speedup looks acceptable for 2 and 3 cores across the board. In this figure, the spike that occurs at 50 rooms is more visible than in the previous one. It seems to be present in each of the numbers of cores, which is strange as one would expect that the peaks would occur at random points if the cause was simply a random variation in runtime. However, I expect that this is just a random occurrence nevertheless.

5.2 Number of events

To test how the program responds to larger numbers of events, the total number of events was increased for different numbers of cores. The rooms were left the same as in the previous test. The results should be somewhat similar to the previous test, but the speedup should be lower as the calculations must all be run in sequence in this case.

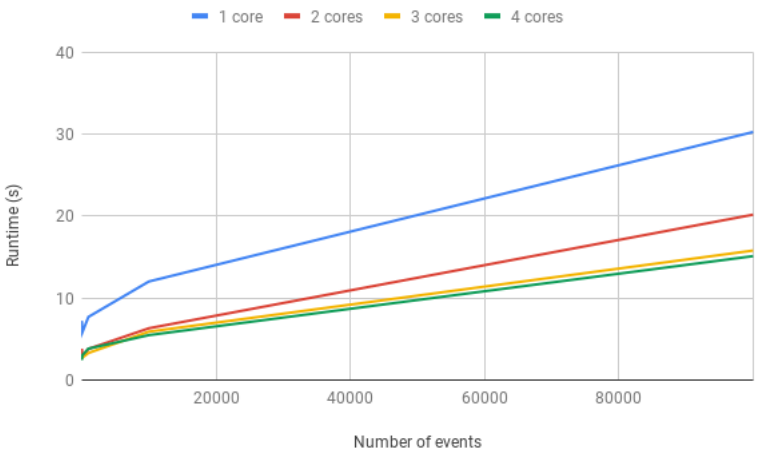

Figure 7: Runtime for different numbers of events

The runtime figure looks almost identical to the one indicating the runtime depending on the number of rooms. One clear difference, however, is that 100000 events can be processed in a shorter time on 1 evaluator than 10 events on 10000 evaluators. Thus, the overhead of creating all of these evaluators may be greater in this simple situation than the benefit of having multiple, completely independent threads.

As expected, the speedup in this case is lower than in the prior test. Whereas the speedup for 3 and 4 cores hovered around 3 when there were many evaluators, the speedup when there are many events is around 2. Since there is no increased parallelization, the speedup remains constant, although the first data point for 4 cores seems to be a slight outlier with a speedup of 3.

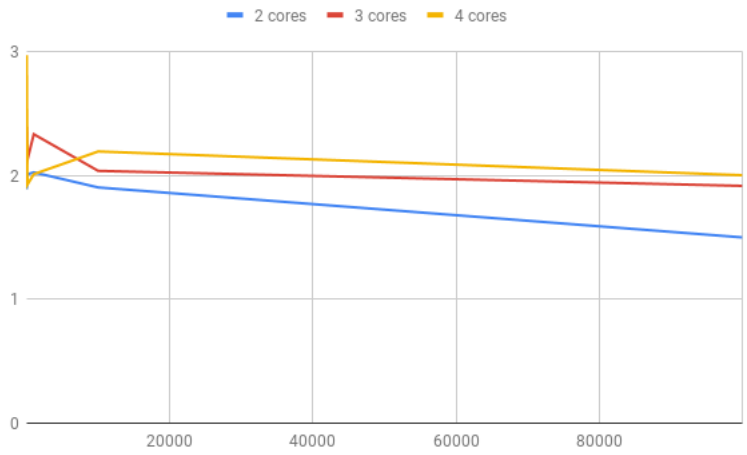

Figure 8: Speedup for different numbers of events

6 Conclusion

For this project, a program was made that can be used to run PDES like simulations for smart buildings. The program is very basic and could use more optimization. Overall, the performance in simple cases looks acceptable. As outlined in the future work section, however, the program is lacking in several ways.

Running simulations of smart buildings using the actor model looks to be an appropriate way of simulating smart buildings to test control programs, if the right data is available in order to create a realistic situation. Smart buildings specifically can be simulated in a simplified way due to the fact that only one type of process, the sensors, must wait. With no method of comparing the data to some other method of simulation, for example an optimistic version, however, it is difficult to know exactly how valuable this architecture is to the simulation of smart buildings.

It is clear from the results, however, that solving for different parts of the model independently increases performance as the speedup in the first test as the number of independent rooms increased quickly reached around 3, higher than the speedup in the case where the number of events was increased. As such, using actors in a conservative approach to synchronization seems to be an efficient method of making simulations of larger smart buildings or grids possible on many cores, although performance could not be tested on a machine with more than 4 cores.

7 Future work

For this project, a program was made that is capable of running a distributed simulation of smart buildings, given some building, a set of user rules, the events that should occur during the time frame and the simulated physical relation between actuator and sensor states. Each of these aspects are required in order to generate realistic results.

The program could still potentially be improved within its original scope. A version of the program could be made that uses optimistic methods, rolling back events whenever necessary. This would render the program much more parallelizable, although not all of the computations performed will be useful due to the rollbacks. It could result in a faster simulation, however, as events that could have been related without ending up being so will be performed simultaneously, unlike in the program as it is.

Another possibility is the optimization of the use of the CSP solver. Currently, a new model is made whenever solving is necessary. A better version of the program would modify the model instead, removing constraints that are no longer active and adding newly active constraints.

The program in its current iteration is very simple. To improve its usability, file IO for rules and simulated buildings would be essential as well as some UI that allows the use of files to run simulations and to show visualizations of generated data. Furthermore, some possibility should exist for users to insert external simulation functions or programs such that the simulation can rely on data and calculations from other programs to determine sensor states.

References

[1] L. Pérez-Lombard, J. Ortiz, and C. Pout, "A review on buildings energy consumption information," *Energy and Buildings*, vol. 40, no. 3, pp. 394 – 398, 2008.

[2] F. Amara, K. Agbossou, A. Cardenas, Y. Dubé, and S. Kelouwani, "Comparison and simulation of building thermal models for effective energy management," *Smart Grid and renewable energy*, vol. 6, no. 04, p. 95, 2015.

[3] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, Oct. 1990.

[4] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *J. ACM*, vol. 40, pp. 304–333, Apr. 1993.

[5] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simulation Modelling Practice and Theory*, vol. 30, pp. 54 – 73, 2013.

[6] V. Degeler and A. Lazovik, "Dynamic constraint satisfaction with space reduction in smart environments," *International Journal on Artificial Intelligence Tools*, vol. 23, p. 1460027, 12 2014.

[7] P. H. Shaikh, N. B. M. Nor, P. Nallagownden, I. Elamvazuthi, and T. Ibrahim, "A review on optimized control systems for building energy and comfort management of smart sustainable buildings," *Renewable and Sustainable Energy Reviews*, vol. 34, pp. 409 – 429, 2014.