The Road to a Building Automation DSL through MDD

Hansen, K. Kontostathis, K., Schmidt, E.

IT-University of Copenhagen, Denmark,
kben@itu.dk, kkon@itu.dk, eker@itu.dk

Abstract. The paper presents a Domain Specific Language (DSL) in the domain of building automation that aims at analyzing, controlling, and optimizing both energy consumption and human comfort in modern buildings. The main contribution is a large DSL built using Eclipse Modeling Framework and Xtext that captures the main concepts in this domain. Requirements were qualitatively elicited from six organizations in Denmark via semi-structured open-ended interviews. The paper describes the elicitation of these requirements, their implementation in a DSL (both abstract syntax and textual concrete syntax), an evaluation of the DSL by modeling parts of the interviewed organizations's buildings, and discusses future work.

1 Introduction

Energy and natural resources are precious commodities. Residential buildings use about 89% of the total energy consumption for space heating, cooling and water heating [9]. Electrical appliances use 11%. Other buildings use 79% of the total energy consumption for space heating, cooling, water heating and lighting [9]. To *manually* control buildings for energy efficiency and optimal human comfort is a time-consuming and inefficient task, if even possible. *"Worldwide, there is no doubt that efficient energy saving is only possible with modern BA*[1] *based on networking in all levels of abstraction."* [8]. Constructing resource efficient buildings makes sense, both from a political and economical perspective.

Consequently, our collective need is the adaptation of buildings to the users and the sensor-perceived environment. This can be achieved by developing governing *policies* (defined as pieces of code) based on input such as semi-static data, dynamic data and sensor input, which control the actuators and thereby leading to the desired building state(s).

Merriam-Webster defines *policy* as "a definite course or method of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions."

Our goal is to facilitate the management of building automation using the concept of a DSL rich and expressive enough to capture concepts in that domain.

The main contribution is a large building automation DSL implemented using EMF [3] and Xtext [6]. Six companies were interviewed regarding their need for building automation *policies*. After analyzing the interviews, we iteratively constructed the DSL

[1] Building Automation

for building automation. We argue by example, that our DSL is rich and expressive enough to capture the requirements put forth in the interviews.

The focal point is the descriptive properties of the DSL, in order to facilitate the construction of policies expressed during the interviews — and not any integration towards existing subsystems like CTS[2], calendars and/or custom made domain specific subsystems. Those systems are outside the scope of this project.

We have tailored our method as follows;

1. Interviews and analysis — we conducted a series of open-ended interviews with people working in, or close, to FM[3] in various danish companies.
2. Design and development — we designed a building automation DSL based on our analysis of the interviews
3. Evaluation — we evaluated the completeness, expressiveness and richness of the DSL by implementing core requirements extracted by the interview with Brüel & Kjær and with Københavns Tekniske Skole since those two companies contained the most advanced behavior put forth during the interviews.

The structure of the paper is as follows. We start by discussing the Related Work (section 2) and compare it with our own. Then we elobarate on the Design and Development (section 4) featuring a comprehensive collection of example DSL code. We go on to explaining our Evaluation (section 5), and it's results in the Discussion (section 6). Threats to Validity (section 7) describes possible risks to our work and how we eliminated them, while in Future Work (section 8) we provide several suggestions for improving the DSL. Last we provide a Conclusion (section 9).

2 Related Work

A lot of research has already been conducted in home and building automation, spanning from low level communication protocols like BACnet, LonWorks and EIB/KNX [11] to full system implementations. In comparison, less progress has been made on user friendly DSL designs, where everyday building concepts are mixed with time constraints and conditional logic for building automation.

In [7], a DSL named SmartScript is developed. Based on a publish/subscribe paradigm it is intended for appliance control, and implemented with a KNX adapter. It allows grouping of devices to make it possible for instances to turn on the light and dim it using the same 'light' concept, even though the light switch and dimmer are two separate electronic components. The language features three types of statements; Action, If and Loop. Specifically, there are two actions; Set and Get. Although it is possible to use the group concept, it does not seem to be possible to define the entire building with rooms based on different types. This results in a long collection of variables, even for small and simple buildings. Moreover, the script seems to be operating on a much lower

[2] Central Tilstandskontrol og Styring (Building Automation and Control System)
[3] Facility Management

abstraction level than our DSL, maybe due to its implementation that seems to have pulled the DSL more into the direction of a GPL[4].

In [10], there is a clear distinction between usage of the domain experts (called the Catalog View) and the domain users (called the Application view). A graphical editor is used when designing an application, by dragging and dropping types into flows that will be orderly executed. However, the user designing the application still has to get familiar with concepts such as *standard functional units*, *FUnitLinks* and *Scenes*. Again, our DSL is at a higher abstraction level. However, the catalog view might be considered as the current state of our meta-model, where the different concepts are defined. Our model only operates on concepts already known to people expected to work with building automation like *Building, Floor, Room, Sensor, Actuator, Schedule, and Policy*. Apart from the obvious differences, our DSL is text-based, versus the graphical editor in Habitation, our flow resides in the combinatory logic derived from time schedules, room types, boolean states and conditional logic specified in the policies. This will become evident in Design and Development (section 4).

In the Google sponsored Home Automation Bus — openHAB [4] it is also possible to specify rules, running on actual implemented hardware. OpenHAB offers a whole suite of implemented protocol standards and functionalities, that includes an Xtext based script interpreter. OpenHAB's rule language is, like most other systems, very low level and does not offer the expressive benefits of a high abstraction DSL.

3 Interviews and Analysis

Since building control resides with FM we found it pertinent to interview company employees working in or close to that field. We conducted open-ended interviews with these companies and documented their content mostly by handwritten notes or in a single case by audio. The main purpose of the interviews was to document;

1. existing governing rules, implicit or explicit.
2. existing and requested, sensors and actuators.
3. requests for new governing rules regardless of their practicality or feasibility.

In order to gather enough material to conceptualize a proper DSL, we interviewed people from the following companies;

Brüel & Kjær supplies integrated solutions for the measurement and analysis of sound and vibration with facilities located in Nærum, Denmark. These facilities total up to an area of approximately 20,000m^2 and are divided into 2 building with 2 floors each;

– Main building is used for office space, calibration of new products, research and development,“university” which is used for seminars and conferences, cafeteria and storage space.
– Repair building is only used for repairs and testing of products.

[4] General Purpose Language

These facilities are daily used by 500 employees for various functionality and only 5 are employed in the FM Department to maintain the buildings. Some of the policies in place are; `LightControl`—in all toilets, `TemperatureControl`—in offices, production, calibration, repair and `HumidityContol`—in production, calibration and testing rooms.

Interviewee Building Maintanance Manager.
Duration 1 hour in total, face to face
Interview Audio Recording

Bygningsstyrelsen has the responsibility for the management of some of Denmark's important state institutions (for example *universities*, *ministries* and *governmental offices*) and for parts of the legislation in that domain. The interview started by asking about the least and the most advanced buildings they manage;

– Even the least advanced buildings feature existing CTS systems, and are primarily used for heating only. No other sensors or actuators might be available.
– The most advanced buildings features many different types of sensors and actuators, still including a CTS system for temperature control and sometimes for other purposes as well.

Interviewee Chief Consultant (Chefkonsulent), Manager
Duration 2 hours in total by phone
Interview Transcribed

IT-University of Copenhagen is an educational institution based in Copenhagen. The facility is one main campus with an approximate area of 19,000m^2. The building has a basement and 5 floors.

– The basement is used for bicycle parking and gym facilities.
– The ground consists of an atrium, canteen, lecture rooms and offices.
– Like the ground floor, the rest of the floors consist of lecture rooms and offices.

Interviewee Building Maintenance Manager.
Duration 1 hour
Interview Transcribed

Københavns Tekniske Skole is an educational institution based in Copenhagen with colleges in various parts of the city. The interview was conducted with an employee in the school at Dragør, approx. 9km outside Copenhagen.

– The basement is used for storage of furniture and disused equipment.
– The ground floor features classrooms, rooms for teachers, kitchens, toilets, hallways, a canteen, and rooms for various technical equipment like computers, photographical equipment etc.
– The top floor contains a hallway with skylights and more classrooms and offices.

Interviewee Værkstedsassistent (Workshop assistant).
Duration 1,5 hours in total by phone
Interview Transcribed

ST Aerospace - Denmark is an independent global company, offering third party aviation maintenance, repair and overhaul. They have facilities in Copenhagen, near the airport, and the building has 5 floors. Details about their building were not discussed, since the management of the building is provided by a third party — however general concepts were extrapolated from the interview.

Interviewee Head of FM.
Duration 0,5 hour by phone
Interview Transcribed

UNI-C is an agency under The Danish Ministry of Children and Education. They promote digital development within the area of children and education. Their primary focus is to increase the use of IT in education, and supporting an effective operation of institutions by using IT. UNI-C has just moved into a refurbished and modern building with 5 floors. It is located close to the central town square (Rådhuspladsen) featuring ventilation system, and automated blinds.

- All the floors, except the roof terrace, contains offices.
- The basement is used for storage.
- 4th floor has a big canteen.
- Some of the floors have conferencing rooms.

Interviewee Kontorfuldmgtig (administrative officer)
Duration 1 hour
Interview Transcribed

Questions were formulated by the team to be used as a base for these interviews and aimed at gathering relevant information for our project. Some of the questions used in the conducted interviews are shown below:

1. How are buildings divided and what is the total floor area?
2. What is the facility used for?
3. How many employees make use of the facility?
4. How many work in FM department?
5. What tools, if any, do you make use of currently?
6. What policies are in place today?
7. Which of them would you like to have implemented?
8. What sensors and actuators are in use today and which others do you find relevant and necessary to have?
9. What other implementations would you consider to make the management of the facilities easier?

While conducting the interviews, we pointed out that practicality or feasibility should not influence the requests for new governing rules. This proved hard for the interviewees. Some interviews were therefore conducted over several sessions, giving the interviewees time to think creatively. We only used natural languages (English and Danish) to give examples of policies to facilitate the discovery of new, relevant ones. This was done to avoid the limitations of technology — like the expressiveness of a programming language and to make communication less rigid. Notes from some of these interviews can be found in Appendix D - Transcribed Interviews.

Finally, this project is solely based on the conceptual requirements extrapolated from the interviews. We use the state policies as the definition on how to design the DSL. The successful grammar parsing therefore constitutes a the working DSL. For example, Brüel & Kjær requested that in case a humidity sensor in the production area drops below a certain value, the alarm in the janitors office should go off. This is implemented and can be seen in fig.7. In Design and Development (section4), we elaborate more on how these interviews contributed to the design of the DSL.

4 Design and Development

Since our complete meta-model [1] is of considerable size, we will in this section explain it using a collection of smaller Ecore diagrams. We also explain the key concepts behind the definition of policies.

By using Eclipse [2], EMF and Xtext we built a domain specific language that is rich enough to express the policies discovered in the analysis. We developed a grammar that is based on our programming experience, and believed to have a consistent forthcoming syntax and conceptual constructs (see Appendix C - Policy Engine DSL (*PeDsl*) - Xtext). Finally, we used our Policy Engine DSL editor to specify buildings and write examples of policies learned from the interviews. The full DSL implementation based on two of the interviews, Brüel & Kjær (see Appendix A - Brüel & Kjær) and Københavns Tekniske Skole (see Appendix B - Københavns Tekniske Skole). The referenced DSL figures are all cropped from screendumps of actual, running Eclipse instances.

4.1 Meta-Models

The following core packages, differentiated in separate Ecore Diagrams can be viewed fully in [1] as a single meta-model. In the subsections below, we show images and explain the build up of the core package as it is in our model.

Building definition In order to specify the building(s) with all their floors and rooms, the core classes has been included in fig.1. They all inherit from NamedElement, making it possible to give them a name than can be used for later referencing.

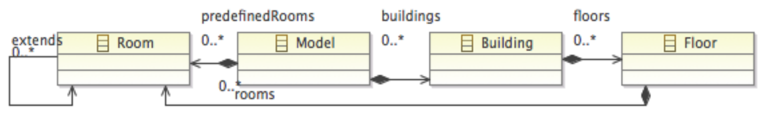


Fig. 1. The meta-model related to the *building definition*. The inherited class NamedElement has been omitted.

Sensors and Actuators In our meta-model, the sensors and actuators used are from the combined analysis of the interviews mentioned in the section above. In the meta-model, we have a total of 9 sensor classes and 6 actuator classes. Sensor and Actuator classes inherit from NamedElement, which gives declared elements a name that later can be referenced in other parts of our DSL. If new types are needed, they should be defined in the meta-model.

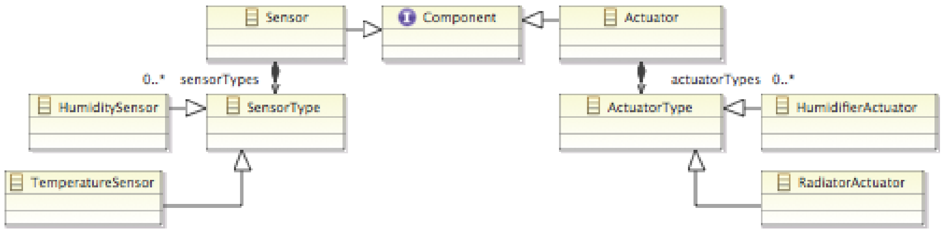


Fig. 2. A fragment of the *sensor* and *actuator* meta-model hierarchy.

Expression Language After the analysis of the interviews we concluded that the needed expression language did not need to be very advanced. We needed boolean variables (*states*) and if-statements with two different types of conditions; the normal if-statement — which operates on sensor types and state instances – and another that operates on a timer (see fig.5). In the body of the if-statement, we can use an expression to reset timers, and set actuator values using instances in specific rooms as shown in the DSL example in fig.7. The core classes are all included in fig.3 below.

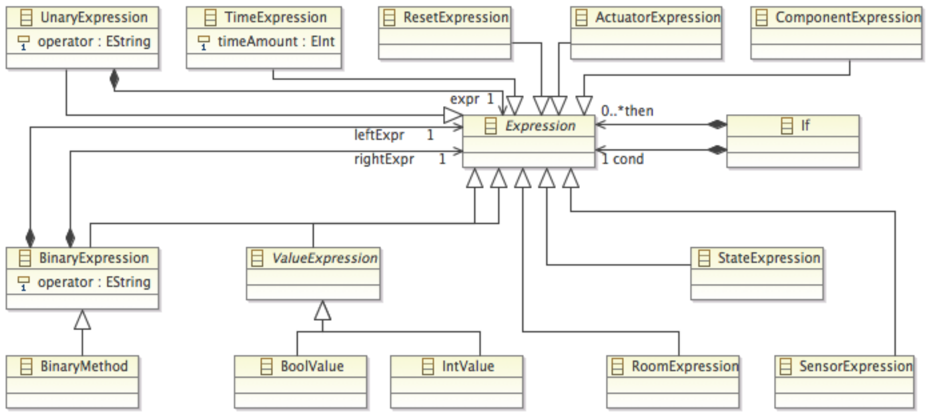


Fig. 3. The meta-model related to the *expression language*. The inherited class NamedElement has been omitted.

Policy definition The policy definition meta-model can be inspected in fig. 4. To define a policy, we make use of actuator types, sensor types and room instances (via `usesRooms`). The implementation of the policy is enhanced by the use of the statements; *if*, *state* and *timer*. The if-statement uses the expression language to define it's behavior. *Schedule* can be defined so that policies are temporally constrained — as can be inspected in fig. 8. All policy definition core classes are displayed in fig.4.

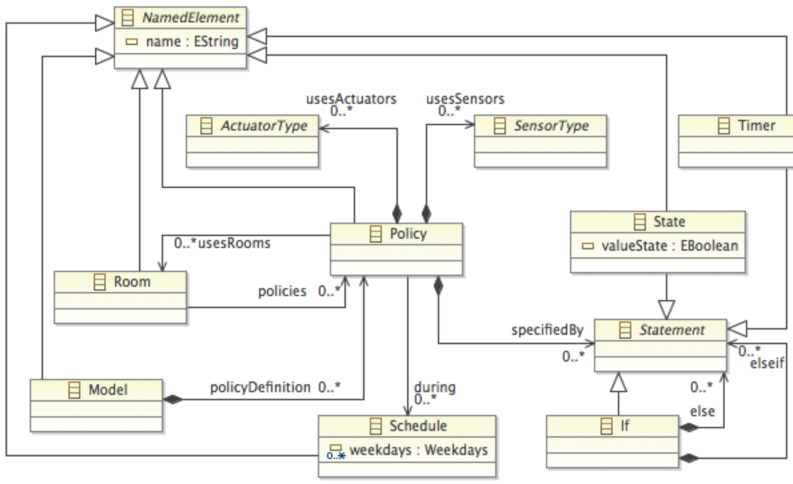


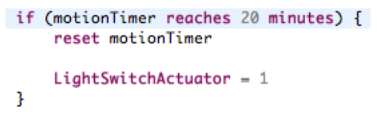
Fig. 4. The meta-model related to the *policy definition*.

4.2 Key Concepts

During the analysis of the interviews, different meta-constructs were defined. It was evident that the *three key concepts* listed below were needed in order to tailor core functionalities discovered.

Time Time has to be an integral part of the DSL, and not just related with the internal policy logic. Several interviewees mentioned concepts like weekdays, weekends, normal working hours, holidays, night, day, morning etc., which clearly shows that Time is necessary. We have extrapolated the need for Time in two different cases;

1. Time conditional expressions are expressions used for determining the flow of a behavior. It is 'if' statements that can react based on a built-in timer function (see fig. 5).

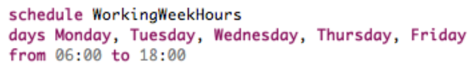


```
if (motionTimer reaches 20 minutes) {
    reset motionTimer

    LightSwitchActuator = 1
}
```

Fig. 5. An example of a *time conditional expression* that evaluates to true if more than 20 minutes have elapsed.

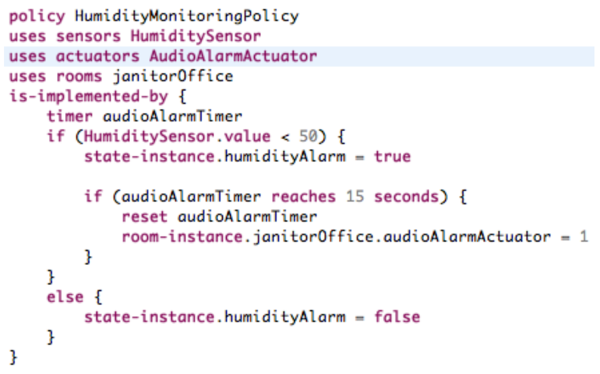
2. Schedules are predefined types representing a timespan (see fig. 6), which later can be attached to a specific behavior where action or no-action takes place. Note that by defining several *schedules*, it is possible to make not only schedules that can overlap in time, but also policies that take over when other policies reach their end.



```
schedule WorkingWeekHours
days Monday, Tuesday, Wednesday, Thursday, Friday
from 06:00 to 18:00
```

Fig. 6. An example of a *schedule*.

Policy Policies define the actual behavior, ie. adjustment of actuators, based on sensor input. In fig. 7, when the `humiditySensor` value drops under 50, the `humidityAlarm` state is set to true. As long as the `humidityAlarm` is true, the `audioAlarmActuator` in the `janitorOffice` will sound every 15 seconds.



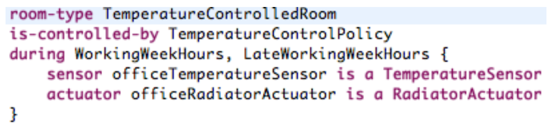
```
policy HumidityMonitoringPolicy
uses sensors HumiditySensor
uses actuators AudioAlarmActuator
uses rooms janitorOffice
is-implemented-by {
    timer audioAlarmTimer
    if (HumiditySensor.value < 50) {
        state-instance.humidityAlarm = true

        if (audioAlarmTimer reaches 15 seconds) {
            reset audioAlarmTimer
            room-instance.janitorOffice.audioAlarmActuator = 1
        }
    }
    else {
        state-instance.humidityAlarm = false
    }
}
```

Fig. 7. An example of *policy* using a single sensor type, and an actuator for a room instance.

Building specification Rooms and room types are part of the complete building specification, with terminology rooted in concepts revolving around buildings, ie. rooms, room types, floors, different sensors and actuators.

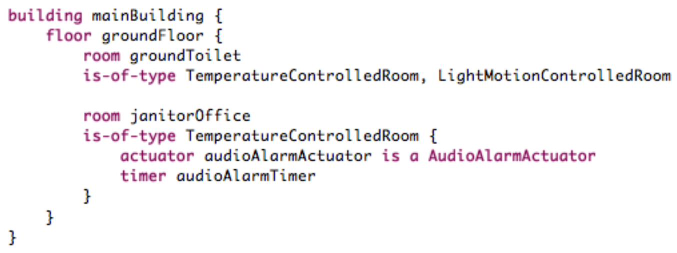
To avoid cluttering the DSL — and confusing the user with an overwhelming amount of declarations of rooms, sensors and actuators — we have designed *room types* (fig. 8) that declare static use of sensors and actuators. Rooms that does not fit into types, can still be internally defined by sensors and actuators on the instance level.



```
room-type TemperatureControlledRoom
is-controlled-by TemperatureControlPolicy
during WorkingWeekHours, LateWorkingWeekHours {
    sensor officeTemperatureSensor is a TemperatureSensor
    actuator officeRadiatorActuator is a RadiatorActuator
}
```

Fig. 8. An example of *room types*.

The *building specification* (fig. 9) is the last piece of the BA puzzle — it helps us analyze the the facilities that requires automation. Using this descriptive concept, we we are able to accurately define the real-world objects as *buildings*, *floors* and *rooms* — as well as implement policies for rooms in specific locations.



```
building mainBuilding {
    floor groundFloor {
        room groundToilet
        is-of-type TemperatureControlledRoom, LightMotionControlledRoom

        room janitorOffice
        is-of-type TemperatureControlledRoom {
            actuator audioAlarmActuator is a AudioAlarmActuator
            timer audioAlarmTimer
        }
    }
}
```

Fig. 9. An example of *building specification*. Note that the *janitorOffice* inherits from a *TemperatureControlledRoom* (which uses a *TemperatureSensor* and a *RadiatorActuator*) but also declares a *AudioAlarmActuator* and a *Timer* instance.

All the sample DSL implementations as showns in the figures above, are possible by the fact that our grammar has a consistent forthcoming syntax and conceptual constructs. It has been tailored to fit the industry by making use of commonly used words in FM, which makes it clean, understandable and easy to use. (see Appendix C - Policy Engine DSL (*PeDsl*) - Xtext).

5 Evaluation

The evaluation of our DSL was conducted iteratively and based on the successful modeling — using the DSL — of buildings and policies identified in the analysis of the interviews with the two organizations; Brüel & Kjær and Københavns Tekniske Skole, mainly because they contain the most advanced requirements for BA. Another method of evaluation would have been to have these organizations evaluate our DSL, but due to time constraints on both our project and in the companies this was not possible.

Our DSL can capture the most important parts of the organization's building specification and can define the policies identified (without involving existing custom subsystems as described in 1).

In Brüel & Kjær, our DSL can capture all parts of the organization's buildings and all the policies identified in the interview. One of the requirements was not implemented because it was outside the scope — *store all the sensor data in a database in order to be further used by a web-based system to visualize the data in the janitor's office*.

In Appendix A - Brüel & Kjær, our DSL manages to capture the building specification as it is; two buildings, each consisting of two floors and each floor of several rooms. We were able to define rooms types and extend rooms to be of those types. The DSL also captures most of the policies identified; *schedules* and *policies* are defined. The policies make use of both actuators and sensors as well as the room types. In the implementation we use the expression language to check sensor values and based on that we set actuator types or the defined instances in specific rooms to a desired state.

In Appendix B - Københavns Tekniske Skole, Københavns Tekniske Skole, just as in the case above, our DSL captured most of the policies and predefined room types as in the interview. In this evaluation we focus more on modeling policies, use of states, timers and room-types.

During an evaluation iteration we realized that some of the sensors and actuators needed to satisfy the implementation were not defined in our meta-model. Prioritizing our time, we decided to turn our attention elsewhere — and this could have been easily fixed by simply adding the new classes in the meta-model, as discussed in Defining Actuator and Sensor types (section 8.5). A missing feature was discovered in a later evaluation iteration of the policies presented in Københavns Tekniske Skole. It would have been optimal to chain together a given policy and it's schedule, instead of the current *during concept* where a bunch of policies is sharing a schedule. This is discussed further in Policy Scheduling (section 8.7). The expression language was also found to be lacking important functionality in the earlier iterations — for example to handle recursive expressions and the negation of expressions — but was later improved and now works as depicted in the DSL examples.

Using our iterative method of evaluation against defined policies from interviews enabled us to find weaknesses, which could be fixed in time, as well as areas that can be further developed in Future Work. (section 8).

6 Discussion

During the design and implementation of our meta-model we made many different design decisions, primarily based on the length and scope of this project and revelations from the interviews. Some of these decisions merit explanations.

6.1 Expression Language

Since the expression language needed for our DSL is straightforward, we chose to implement it ourselves. We also looked at basing our expression language on the functionality provided by XBase [5], however the current state of XBase seems far from a stable product. Since our timeframe was rather limited, we did not want to use our time with the implementation details of XBase. We expect XBase to further develop as time goes, and later it might be an excellent choice in regards to the expression language.

6.2 Class Use versus Instance Use

As can be observed in fig.7 in Design and Development we rely heavily on class use instead of instance use. Consider the DSL sentence "uses sensors `TemperatureSensor`" — it uses a `TemperatureSensor` as a class, and not as an instance. We do not find this anymore uncommon than using, for example, static functions in Java or passing a Class to a Java generic. It is simply a way to offer the user flexible functionality in a non-complicated way. Imagine that we had to define a 10 story building, room by room, sensor by sensor, actuator by actuator and so forth. The resulting DSL source would quickly become littered with instances, which probably would need to carry the name of the room in it's name in order for the user to keep track of them. An example could be the fictive sensor *buildingFactory3rdFloorRoomNextToKitchenTemperatureSensor*. By using classes directly, we can define room types constructed from both their common sensors and actuators, but also on their need for governing policies. It is still possible to use instances, as also shown in fig.7 in Design and Development — *janitorOffice*, if we for example, need to activate *a single* actuator based on input from several room sensors. Aspects of this is addressed in Future Work (see section 8.1).

7 Threats to Validity

7.1 Representativeness

The main threat to the validity of our developed DSL is that the interviews were restricted only to the six danish companies as mentioned in Interviews and Analysis (section 3). We do not know if the educed requirements would be representative enough for a larger base of customers as the derived work is for these particular companies. On the other hand we argue that — based on the organizations, buildings and policies described in the interviews — our DSL most probably would be rich and expressive enough to satisfy a large group of companies.

7.2 Assessment of our own work

Another risk to the validity of our project is that we evaluated our own work — though based on actual requirements from the interviews — which makes the evaluation biased. In a more proper setup it would have been ideal to ask the interviewees to evaluate the DSL but due to time constraints (both on our part and the companies involved), this was infeasible. As a consequence, we decided to tailor the evaluation on core requirements (requirements — elicited by us — that seemed advanced and were reoccurring amongst interviewees) and implement those using our DSL.

8 Future Work

8.1 Collections

When managing even more complex buildings, it might be necessary to have collections of instances. This could be simple arrays, lists or something like the generics of Java. It might prove difficult to convey the concept to non-programmers, so the actual syntax should be defined together with future users.

8.2 Graphical Editor for policy definition

Our DSL editor is text based and we have decided to use concepts and words commonly used in the industry *(FM)*. The concepts are structured and used in the DSL so they offer a literal description of the meaning of the combined keywords. However, one needs to have some programming knowledge in order to understand — and be able to use — the expression language. Therefore, our DSL could be improved by developing a graphical editor in order to abstract the expression language and consequently make it simpler for users to use.

8.3 Integration with existing systems

As stated in the Introduction(section1), our focal point in this project is not on subsystem integration. We started out by modeling some of the external systems mentioned by our interviewees — but as we found more and more of those subsystems we decided that they are outside the scope of this paper. As all of the interviewees use CTS systems, it is recommended to persue integration to this system type in future work. Generalization towards this subsystem is in high demand, since it represents a potential vendor lock in trap, especially with regards to service.

8.4 Enhanced Code Completion

Code completion is of great importance to our language because it really helps the user by guiding her with all possible options. Right now, our DSL has working code completion but in the future we would like to restrict code completion to respect code scopes in greater detail. For example if a user while defining a policy uses `TemperatureSensor` and `RadiatorActuator`, she gets a list of all `SensorTypes` and `ActuatorTypes` present in our DSL. In a future revision the scope will be respected and only `TemperatureSensor` and `RadiatorActuator` will be presented as valid options in that particular policy.

8.5 Defining Actuator and Sensor types

During the Evaluation (section5) we came to realize that some of the sensors and actuators needed to satisfy the requirements were not implemented in the DSL. Some were introduced iterativly, but this still does not give us the flexibility to define policies using actuators and sensors that are not already available in our DSL. Future work in this area might include some modification of our meta-model in order to accommodate runtime definitions of actuators and sensors - like we do with *room types*. This will enhance the DSL's usability and flexibility and help make the DSL more adaptable.

8.6 Prioritization

Prioritization of policies should be introduced in future versions of our language. Higher priorities could be given to policies handling situations like fire alarms for example. At the moment we imagine the DSL running in a loop that calls all active (determined using the *schedules*) policies in every iteration, making the timespan between policy executions linear with the amount of active policies, assuming they are of same size and complexity. This could instead be based on a prioritization of room, room-type or policies adding extra flexibility and fine grained control to the language. For instance, specify that some policies should only run once in an hour, and others appear that they are handled instantaneous.

8.7 Policy Scheduling

In its current state, our DSL allows us to determine when the policies should be executed by using the already defined schedules. One or more policies can be executed based on one or more schedules. In the case we have several governing policies that run on different schedules, our DSL does not allow us to assign a specific schedule to a single governing policy. An interesting improvement that could increase our DSL's flexibility would be the ability to specify several schedules for each of the governing policies.

9 Conclusion

Even though the domain of building automation is large, and some existing work is using MDD concepts, no high-level DSL abstraction exists. Our DSL, that is the main contribution to the domain of building automation, addresses this gap.

We have conducted six well-prepared interviews with different danish companies to examine some of their needs with regards to building automation. We focused on *policies* — building automation governing pieces of functionality, specifically what the interviewees already are using and what they would like to see implemented if nothing were impractical or infeasible. Abstract and concrete syntax was iteratively designed and implemented, using an analysis of the interviews as a foundation for requirements. Both were equally refined to have a simplistic, conceivable and professional look.

Our DSL is designed to be a useful tool to anyone residing in facility management or doing building automation. It was designed using already familiar concepts as keywords, such as *policies*, *room*, *floor*, placing the language at a higher abstraction level than existing technologies — and thereby introducing novelty in the domain. The language contains a good amount of flexibility, both in regards to the structure of the automation, the temporal aspect and the possibility to define *room types*. We argue that this increase adaptability and manageability. Great care has been taken to implement the meta-model in such a way that good auto-completion is offered the user, leading to fast, easy and infallible written policies.

The DSL was evaluated based on what was achievable using the DSL in regards to core functionalities extrapolated during the analysis of the interviews. We show that the important core functionalities are achievable using our DSL and argue that our DSL constitutes a novel perspective on building automation. With further improvement, our policy engine has the potential to significantly contribute to the domain of building management.

References

1. The complete building automation dsl ecore diagram. `https://github.com/rugbroed/BPE/blob/Version-0.9/documents/ecore-complete-diagram.png`. Online; accessed 14-May-2013.
2. Eclipse home. `http://www.eclipse.org/`. Online; accessed 20-April-2013.
3. Eclipse modeling framework. `http://www.eclipse.org/modeling/emf/`. Online; accessed 20-April-2013.
4. Home automation bus (openhab). `https://code.google.com/p/openhab`. Online; accessed 22-April-2013.
5. Xbase wiki entry at eclipse.org. `http://wiki.eclipse.org/Xbase`. Online; accessed 20-April-2013.
6. Xtext home. `http://www.eclipse.org/Xtext/`. Online; accessed 20-April-2013.
7. Diego Adolf, Ettore Ferranti, and Stephan Koch. Smartscript - a domain-specific language for appliance control in smart grids. In *Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on*, pages 465–470, 2012.
8. Dietmar Dietrich, Dietmar Bruckner, Gerhard Zucker, and Peter Palensky. Communication and computation in buildings: A short introduction and overview. *IEEE Transactions on Industrial Electronics*, 57(11):3577–3584, 2010.
9. Rod Janssen. Towards energy efficient buildings in europe. `http://www.alexandra-dock-project.com/media/22040/towards%20energy%20efficient%20buildings%20in%20europe.pdf`. Online; accessed 13-May-2013.
10. M. Jimenez, F. Rosique, P. Sanchez, B. Alvarez, and A. Iborra. Habitation: A domain-specific language for home automation. *IEEE Software*, 26(4):30–38, 2009.
11. W. Kastner, G. Neugschwandtner, S. Soucek, and H.M. Newmann. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178–1203, 2005.

Appendix A - Brüel & Kjær

```
BruelOgKjaer {

        state humidityAlarm = false
        state fireAlarm = false

        policy TemperatureControlPolicy
        uses sensors TemperatureSensor
        uses actuators RadiatorActuator
        is-implemented-by {
                if (TemperatureSensor.value <= 18) {
                        RadiatorActuator.setValue = 1
                }
                else {
                        if (TemperatureSensor.value >= 23) {
                                RadiatorActuator.setValue = 0

                        }
                }
        }

        policy SkylightWindowControlPolicy
        uses sensors TemperatureSensor ,RainSensor
        uses actuators WindowActuator
        is-implemented-by {
                state skylightOpen
                if (RainSensor.value = 1) {
                        WindowActuator.setValue = 0
                        state-instance.skylightOpen = false
                }
                else {
                        if (TemperatureSensor.value >= 20 &&
                            state-instance.skylightOpen = false &&
                            RainSensor.value = 0) {
                                WindowActuator.setValue = 1
                                state-instance.skylightOpen = true
                        }
                }
        }

        policy HumidityMonitoringPolicy
        uses sensors HumiditySensor
        uses rooms janitorOffice
        is-implemented-by {
                timer audioAlarmTimer

                if (HumiditySensor.value < 50) {
                        state-instance.humidityAlarm = true

                        if (audioAlarmTimer reaches 15 seconds) {
                                reset audioAlarmTimer
                                room-instance.janitorOffice.audioAlarmActuator = 1
                        }
                }
                else {
                        state-instance.humidityAlarm = false
                }
        }
```

```
policy LightMotionPolicy
uses sensors MotionSensor
uses actuators LightSwitchActuator
is-implemented-by {
        timer motionTimer

        if (MotionSensor.value = 1) {
                reset motionTimer
        }
        else {
                if (motionTimer reaches 20 seconds) {
                        reset motionTimer

                        LightSwitchActuator.setValue = 1
                }
        }
}

schedule WorkingWeekHours
days Monday, Tuesday, Wednesday, Thursday, Friday
from 06:00 to 18:00

schedule LateWorkingWeekHours
days Monday, Tuesday, Wednesday, Thursday, Friday
from 18:00 to 22:00

schedule LockdownWeekHours
days Monday, Tuesday, Wednesday, Thursday, Friday
from 22:00 to 06:00

schedule LockdownWeekendHours
days Saturday, Sunday
from 00:00 to 00:00

room-type TemperatureControlledRoom
is-governed-by TemperatureControlPolicy
during WorkingWeekHours, LateWorkingWeekHours {
        sensor temperatureSensor is a TemperatureSensor
        actuator radiatorActuator is a RadiatorActuator
}

room-type HumidityControlledRoom
is-governed-by HumidityMonitoringPolicy
during-always {
        sensor humidifierSensor is a HumiditySensor
        actuator humidifierActuator is a HumidifierActuator
}

room-type LightMotionControlledRoom
is-governed-by LightMotionPolicy
during-always {
        sensor motionSensor is a MotionSensor
        actuator lightSwitchActuator is a LightSwitchActuator
}

room-type HumidityAndTemperatureControlledRoom
is-of-type TemperatureControlledRoom, HumidityControlledRoom
```

```
        building mainBuilding {
                floor groundFloor {
                        room groundToilet
                        is-of-type TemperatureControlledRoom, LightMotionControlledRoom

                        room janitorOffice
                        is-of-type TemperatureControlledRoom {
                                actuator audioAlarmActuator is a AudioAlarmActuator
                                timer audioAlarmTimer
                        }

                        room office
                        is-of-type TemperatureControlledRoom, LightMotionControlledRoom

                        room calibrationRoom
                        is-of-type HumidityAndTemperatureControlledRoom

                        room cleanRoom
                        is-of-type HumidityAndTemperatureControlledRoom
                }

                floor topFloor {
                        room university
                        is-of-type TemperatureControlledRoom, LightMotionControlledRoom
                        is-governed-by SkylightWindowControlPolicy during WorkingWeekHours

                        room topToilet
                        is-of-type TemperatureControlledRoom, LightMotionControlledRoom
                }
        }

        building repairBuilding {
                floor groundFloor {
                        room repairRoom
                        is-of-type HumidityAndTemperatureControlledRoom
                }
        }
}
```

Appendix B - Københavns Tekniske Skole

```
KobenhavnsTekniskeSkole {
       state humidityAlarm = false
       state fireAlarm = false
       state lockDown = false
       state equipmentState = false
       state computerOn = false

       policy ComputerShutDownPolicy
       uses actuators ElectricalSwitchActuator
       uses rooms WorkStation
       is-implemented-by {
              if (!state-instance.computerOn) {
                     state-instance.computerOn = true
                     ElectricalSwitchActuator.setValue = true
              }
       }

       policy SkyLightWindowControlPolicy
       uses sensors RainSensor
       uses actuators WindowActuator
       is-implemented-by {
              state isRaining = false
              if (RainSensor.value = true){
                     state-instance.isRaining = true
                     WindowActuator.setValue = false
              }
       }

       policy WinterHeatingPolicy
       uses sensors TemperatureSensor
       uses actuators WindowActuator, RadiatorActuator
       uses rooms HeatControlledRoom
       is-implemented-by {
              if (TemperatureSensor.value <= 18) {
                     WindowActuator.setValue = false
                     RadiatorActuator.setValue = 9
              } else {
                     if (TemperatureSensor.value >= 25) {
                            WindowActuator.setValue = true
                            RadiatorActuator.setValue = 0
                     }
              }

       }

       policy RoomHeatingPolicy
       uses sensors TemperatureSensor, MotionSensor, InfraredLightSensor
       uses actuators RadiatorActuator
       uses rooms HeatControlledRoom
       is-implemented-by {

              state roomInUse = false
              if (MotionSensor.value = true || InfraredLightSensor.value = true) {
                     state-instance.roomInUse = true
                     if ( TemperatureSensor.value <= 18) {
                            RadiatorActuator.setValue = 9
                     }
              }
```

```
        else {
                state-instance.roomInUse = false
        }
        if ( state-instance.roomInUse = false && TemperatureSensor.value >= 10) {
                RadiatorActuator.setValue = 0
        }
}

policy UtilityUsageControlPolicy
uses sensors MotionSensor, InfraredLightSensor
uses actuators ElectricalSwitchActuator, WaterValveActuator, GasValveActuator
uses rooms ClosedRoom
is-implemented-by {
        state utilityClosed = false
        if (MotionSensor.value = false || InfraredLightSensor.value = false)  {
                state-instance.utilityClosed = true
                ElectricalSwitchActuator.setValue = false
                WaterValveActuator.setValue = false
                GasValveActuator.setValue = false
        }
}

policy CentralAccessControlPolicy
uses sensors MotionSensor, InfraredLightSensor
uses actuators DoorActuator, AudioAlarmActuator
is-implemented-by {
        if ( state-instance.lockDown ) {
                DoorActuator.setValue = false
                if  (MotionSensor.value = true || InfraredLightSensor.value = true){
                        AudioAlarmActuator.setValue = true
                }
        }
}

policy CompleteLockdownPolicy
uses actuators LightSwitchActuator, WindowActuator, DoorActuator
uses rooms ClosedRoom
is-implemented-by {
        state isSecure = false

        if ( !state-instance.lockDown) {
                state-instance.isSecure = true
                state-instance.lockDown = true
                LightSwitchActuator.setValue = false
                WindowActuator.setValue = false
                DoorActuator.setValue = false
        } else {
                //send msg to person on duty
        }
}
```

```
policy NaturalHeatControlPolicy
uses sensors TemperatureSensor
uses actuators RadiatorActuator, WindowActuator
uses rooms HeatControlledRoom
is-implemented-by {
       state warmOutside

       if ( TemperatureSensor.value > 25 ) {
              state-instance.warmOutside = true
              WindowActuator.setValue = true
              RadiatorActuator.setValue = false
       }
       else {
              if (TemperatureSensor.value < 15) {
                     state-instance.warmOutside = false
                     WindowActuator.setValue = false
                     RadiatorActuator.setValue = true
              }
       }
}

policy HallwayTemperatureControlPolicy
uses sensors TemperatureSensor, RainSensor
uses actuators RadiatorActuator, WindowActuator
uses rooms Hallway
is-implemented-by {
       if ( TemperatureSensor.value > 25 ) {
              WindowActuator.setValue = true
       }
       if ( RainSensor.value = true ) {
              WindowActuator.setValue = false
       }
}

policy EquipmentShutDownPolicy
uses actuators ElectricalSwitchActuator
uses rooms LectureRoom
is-implemented-by {
       if (state-instance.computerOn != true) {
              state-instance.equipmentState = true
              ElectricalSwitchActuator.setValue = true
       } else {
              state-instance.equipmentState = false
       }
}

policy VentilationControlPolicy
uses sensors TemperatureSensor
uses actuators RadiatorActuator, HumidifierActuator
is-implemented-by {
       if ( TemperatureSensor.value > 25 ) {
              HumidifierActuator.setValue = true
              RadiatorActuator.setValue = false
       }
}
```

```
        schedule WorkingWeekHours
        days Monday, Tuesday, Wednesday, Thursday, Friday
        from 06:00 to 18:00

        schedule LateWorkingWeekHours
        days Monday, Tuesday, Wednesday, Thursday, Friday
        from 18:00 to 22:00


        schedule LockdownWeekHours
        days Monday, Tuesday, Wednesday, Thursday, Friday
        from 22:00 to 06:00

        schedule LockdownWeekendHours
        days Saturday, Sunday
        from 00:00 to 00:00

        room-type WorkStation
        is-governed-by ComputerShutDownPolicy
        during LockdownWeekHours, LockdownWeekendHours

        room-type ClosedRoom
        is-governed-by CompleteLockdownPolicy, UtilityUsageControlPolicy
        during LockdownWeekHours, LockdownWeekendHours

        room-type HeatControlledRoom
        is-governed-by WinterHeatingPolicy, NaturalHeatControlPolicy, RoomHeatingPolicy
        during WorkingWeekHours

        room-type VentilationControlledRoom
        is-governed-by VentilationControlPolicy
        during WorkingWeekHours

        room-type Hallway
        is-governed-by HallwayTemperatureControlPolicy, NaturalHeatControlPolicy
        during-always

        room-type LectureRoom
        is-governed-by EquipmentShutDownPolicy
        during LockdownWeekHours, LockdownWeekendHours

}
```

Appendix C - Policy Engine DSL (*PeDsl*) - Xtext

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

import "platform:/resource/dk.itu.mdd.policyengine/model/PolicyEngine.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model returns Model:
	{Model}
	name=EString
	'{'
		(stateDefinition+=State (stateDefinition+=State)*)?
		(timers+=Timer (timers+=Timer)*)?
		(policyDefinition+=Policy (policyDefinition+=Policy)*)?
		(schedules+=Schedule (schedules+=Schedule)*)?
		('room-type' predefinedRooms+=Room)*
		(buildings+=Building (buildings+=Building)*)?
	'}';

ActuatorType returns ActuatorType:
	LightSwitchActuator | HumidifierActuator | DoorActuator |   WindowActuator |
	RadiatorActuator | AudioAlarmActuator | ElectricalSwitchActuator |
	WaterValveActuator | GasValveActuator;

SensorType returns SensorType:
	MotionSensor | TemperatureSensor | RainSensor | TouchSensor |      LightSensor |
	SmokeSensor | CO2Sensor | InfraredLightSensor | HumiditySensor | DoorSensor |
	WindowSensor;

Statement returns Statement:
	State | Timer | If;

Expression returns Expression:
	Conjunction({BinaryExpression.leftExpr=current}
			operator=('||')
			rightExpr=Conjunction )*;

Conjunction returns Expression:
	Comparison({ BinaryExpression.leftExpr=current}
			operator=('&&')
			rightExpr=Comparison )* ;

Comparison returns Expression:
	Primary({BinaryExpression.leftExpr=current}
		operator=('='|'!='|'<='|'<'|'>='|'>')
		rightExpr=(Primary | ValueExpression) )* ;

RoomInstance returns Expression:
	RoomExpression({BinaryMethod.leftExpr=current}
			operator=('.')
			rightExpr=ComponentInstance )* ;

ComponentInstance returns Expression:
	ComponentExpression({BinaryMethod.leftExpr=current}
				operator=('=')
				rightExpr=ValueExpression )* ;



SetStateActuator returns Expression:
       SetValue({BinaryMethod.leftExpr=current}
                 operator=('=')
                 rightExpr=ValueExpression )* ;

Primary returns Expression:
       UnaryExpression | '(' Expression ')' | TimeExpression | StateExpression  |
       SensorExpression;

Condition returns Expression:
       '(' Expression ')'
;

SetValue returns Expression:
       ActuatorExpression | StateExpression
;

Then returns Expression:
       ResetExpression | SetStateActuator | RoomInstance;

UnaryExpression returns Expression:
       {UnaryExpression} operator=('!')expr= Primary;

ValueExpression returns Expression:
               IntValue  | BoolValue;

Component returns Component:
       Sensor | Actuator;

EString returns ecore::EString:
       STRING | ID;

Building returns Building:
       {Building}
       'building'
       name=EString
       '{'
              (accessControl=AccessControl)?
              (timers+=Timer (timers+=Timer)*)?
              (floors+=Floor (floors+=Floor)*)?
       '}';

Room returns Room:
       {Room}
       name=EString
       ('is-of-type' extends+=[Room] (','extends+=[Room])*)?

       ('is-governed-by' policies+=[Policy] (',' policies+=[Policy])*
              ('during' during+=[Schedule] (',' during+=[Schedule])* | 'during-always'))?
       ('{'
              (declareSensor+=Sensor (declareSensor+=Sensor)*)?
              (declareActuator+=Actuator (declareActuator+=Actuator)*)?
              (timers+=Timer (timers+=Timer)*)?
       '}')?
       ;


Policy returns Policy:
        {Policy}
        'policy'
        name=EString
        ('uses sensors' (usesSensors+=SensorType (',' usesSensors+=SensorType)*)?)?
        ('uses actuators' (usesActuators+=ActuatorType (',' usesActuators
        +=ActuatorType)*)?)?
        ('uses rooms' usesRooms+=[Room] (',' usesRooms+=[Room])* )?
        'is-implemented-by' '{' specifiedBy+=Statement (specifiedBy+=Statement)* '}'
        ;

State returns State:
        {State}
        'state' name=EString (valueState?= '=' EBoolean)?
        ;

Timer returns Timer:
        {Timer}
        'timer' name=EString;

Schedule returns Schedule:
        {Schedule}
        'schedule'
        name=EString
                ('days' weekdays+=Weekdays ( "," weekdays+=Weekdays)*)?
                ('from' from=Time 'to' to=Time)?
        ;

AccessControl returns AccessControl:
        {AccessControl}
        'AccessControl'
        '{'
                ('accessControlSensors' '{' accessControlSensors+=SensorType ( ","
                accessControlSensors+=SensorType)* '}' )?
                ('accessControlDoorLockActuator' '{' accessControlDoorLockActuator
        +=DoorActuator ( "," accessControlDoorLockActuator+=DoorActuator)* '}' )?
        '}';

Floor returns Floor:
        {Floor}
        'floor'
        name=EString
        '{'
                ('room' rooms+=Room ( 'room' rooms+=Room)*)?
        '}';

TemperatureSensor returns TemperatureSensor:
        {TemperatureSensor}
        'TemperatureSensor'
        ;

Actuator returns Actuator:
        {Actuator}
        'actuator' name=EString
        'is a' actuatorTypes+=ActuatorType ( "," actuatorTypes+=ActuatorType)*
        ;


LightSwitchActuator returns LightSwitchActuator:
        {LightSwitchActuator}
        'LightSwitchActuator'
        ;

WindowActuator returns WindowActuator:
        {WindowActuator}
        'WindowActuator'
        ;

HumidifierActuator returns HumidifierActuator:
        {HumidifierActuator}
        'HumidifierActuator'
        ;

DoorActuator returns DoorActuator:
        {DoorActuator}
        'DoorActuator'
        ;

RadiatorActuator returns RadiatorActuator:
        {RadiatorActuator}
        'RadiatorActuator'
        ;

AudioAlarmActuator returns AudioAlarmActuator:
        {AudioAlarmActuator}
        'AudioAlarmActuator'
        ;

ElectricalSwitchActuator returns ElectricalSwitchActuator:
        {ElectricalSwitchActuator}
        'ElectricalSwitchActuator'
        ;

WaterValveActuator returns WaterValveActuator:
        {WaterValveActuator}
        'WaterValveActuator'
        ;

GasValveActuator returns GasValveActuator:
        {GasValveActuator}
        'GasValveActuator'
        ;

MotionSensor returns MotionSensor:
        {MotionSensor}
        'MotionSensor'
        ;

DoorSensor returns DoorSensor:
        {DoorSensor}
        'DoorSensor'
        ;

WindowSensor returns WindowSensor:
        {WindowSensor}
        'WindowSensor'
        ;
```

```
RainSensor returns RainSensor:
	{RainSensor}
	'RainSensor'
	;

TouchSensor returns TouchSensor:
	{TouchSensor}
	'TouchSensor'
	;

LightSensor returns LightSensor:
	{LightSensor}
	'LightSensor'
	;

SmokeSensor returns SmokeSensor:
	{SmokeSensor}
	'SmokeSensor'
	;

CO2Sensor returns CO2Sensor:
	{CO2Sensor}
	'CO2Sensor'
	;

InfraredLightSensor returns InfraredLightSensor:
	{InfraredLightSensor}
	'InfraredLightSensor'
	;

HumiditySensor returns HumiditySensor:
	{HumiditySensor}
	'HumiditySensor'
	;

Sensor returns Sensor:
	{Sensor}
	'sensor'
	name=EString
	('is a' sensorTypes+=SensorType ( "," sensorTypes+=SensorType)*)?
	;

If returns If:
	'if' cond=Condition
	'{'
		(then+=Then (then+=Then)*)?
		(elseif+=Statement (elseif+=Statement)*)?
	'}'('else' '{'
		(then+=Then (then+=Then)*)?
		(else+=Statement (else+=Statement)*)? '}'
	)?;

TimeExpression returns TimeExpression:
	time=[Timer] 'reaches' timeAmount=EInt ('seconds' | 'minutes' | 'hours' |
'days' );

ResetExpression returns ResetExpression:
	'reset'  reset=[Timer];

ComponentExpression returns ComponentExpression:
	{ComponentExpression}
		(instance=[Component])?;

StateExpression returns StateExpression:
	{StateExpression}
	'state-instance''.'
		(defineState=[State])?
	;

RoomExpression returns RoomExpression:
	{RoomExpression}
	'room-instance''.'
		(roomInstance=[Room])?
	;

SensorExpression returns SensorExpression:
	{SensorExpression}
		(sen=SensorType'.''value')?
	;

ActuatorExpression returns ActuatorExpression:
	{ActuatorExpression}
		(act=ActuatorType)?
		'.''setValue'
	;

BoolValue returns BoolValue:
	{BoolValue}
	EBoolean
	;

IntValue returns IntValue:
	{IntValue}
	EInt
	;

EInt returns ecore::EInt:
	'-'? INT;

EBoolean returns ecore::EBoolean:
	'true' | 'false';

enum Weekdays:
	MONDAY = 'Monday' | TUESDAY = 'Tuesday' | WEDNESDAY = 'Wednesday' | THURSDAY =
'Thursday' | FRIDAY = 'Friday' | SATURDAY = 'Saturday' | SUNDAY = 'Sunday'
	;

Time returns Time:
	{Time}
		(hours=EShort ':' minutes=EShort)?
	;

EShort returns ecore::EShort:
	'-'? INT;
```

Appendix D - Transcribed Interviews

1. <u>Manager in FM @ Brüel og Kjær</u>

Existing policies:

- Smoke/fire detectors to set off fire alarm and start sprinklers.
- VideoSurveillance (time and weekday based): Records video.
- Access Control on all external doors and in the calibration and clean rooms.
- In the Calibration and Clean rooms air quality (constant) environment is maintained using humidifiers.
- MotionSensors are used in all toilets to turn lights on and off.

New policies:

- The skylight windows can be opened manually by a switch on a nearby wall. Since there is no rain sensor on the window, if forgotten, water damage can be caused. (If it rains outside, then close the window. When workday is over close window.)

- Room radiators have thermostats that senses the ambient temperature and adjusts the radiator accordingly. During winter, workers might find the room 'stuffy' and opens some windows, without bothering to turn down the radiators. The thermostat treats this as the room is getting colder, and turns up the heat. This is a huge waste and very bad for human comfort. (If the radiators are on, and windows are opened then turn off radiators.)

- Windows should be automatically opened based on inner room climate. (temperature, air quality)

- All offices, hallways, university should have motion sensors to control lights.

- Use light sensors to get natural light in the day time and use that to regulate brightness.

- Monitoring of clean rooms and calibration room. if temperature, humidity goes over the operating temperature, set off alarm in the janitors office and send sms/msg to person on duty.

- Log all data in a repository which can used to feed a visual monitoring system and also for easy problem backtrace.

- The central AccessControl system has actuators on only on all door to the outside. (If CLOSING_TIME then SecurityDoors-->Close. If cannot close door, then send sms with alarm to the staff so they can remove obstacle.)

- If work day is over, then enumerate through all windows, skylights and exterior doors. If any of them are not securely closed and locked, then send an accumulated list to the staff.

- Generally it would be nice to have rules/policies working not only by week/time. should also set specific rules for specific rooms. clean room - people work past work hours sometimes.

- The existing control system (CTS) for ventilation is hard to understand. The GUI is made like an electrical diagram, and is not user friendly. Moreover, the temperatures shown in the program is from sensors in the ventilation ducts, not in the room themselves, making it very hard to regulate them, since you'll have to change the settings and then wait for 10-15 minutes. Then you'll have to go to the room and manually hold a thermometer before going back and repeat this.

2. <u>Employee in FM @ Københavns Tekniske Skole</u>

Existing policies:

- AccessControl (time and weekday based): Opens and closes the gate to the school. Can be manually re-programmed and also de-activated. Allows for holydays.

- VideoSurveillance (time and weekday based): Records video.

- RemoteWorkStation (manually): An IT employee can remotely shut off computers, if people forget to turn them off before they leave.
- MotionSensors are used which can be annoying, especially when the light turns off in the middle of a parental meeting, and they have to flail around with their arms to see each other. Should be based on week/time and maybe using Infrared instead of motion.

New policies:

- RemoteWorkStation (automatically): Should be able to set time and weekday for the computers, so they shut off by themselves.

- Today skylight windows can be opened manually by a switch on a nearby wall. There is curtains underneath them, which can also be controlled by a nearby switch. However, people sometimes forget to close a the window, because the curtain is hiding the window. Since there is no rain sensor on the window, the curtain gets ruined by the water and needs replacing. (If it rains outside, then close the window. When school closes, roll back skylight curtains.)

- Room radiators have thermostats that senses the ambient temperature and adjusts the radiator accordingly. During winter, students might find the room 'stuffy' and opens some windows, without bothering to turn down the radiators. The thermostat treats this as the rool is getting colder, and turns up the heat. This is a huge waste and very bad for human comfort. (If the radiators are on, and windows are opened then turn down radiators.)

- By exchanging the room raditor thermostats with centrally controlled temperature sensors and radiator actuators, it will be possible to make a finer grained control of the rooms. Individual room temperature might be possible, using motion/infrared sensors indicating if the room is in use or not. If not in use the heat can be driven down. Should also be possible based on week/time.

- There are gauges on the three main resources in the school; electricity, gas and water. Sometimes, especially after weekends or holidays, an abnormal large usage of water can be registered. Someone might be stealing water for carwashes, or similar. There is a need to shut down the supply if a usage is too high. Should work based on week/time.

- The central AccessControl system has actuators on only a few doors. However, if someone has but in place an obstacle, no notice is given to the people working at the scool, meating that they provide false security. (If CLOSING_TIME then SecurityDoors-->Close. If cannot close door, then send sms with alarm to the staff so they can remove obstacle.)

- If the school is about to close, then enumerate through all windows, skylights and exterior doors. If any of them are not securely closed and locked, then send an accumulated list to the staff.

- Generally it would be nice to have rules/policies working not only by week/time, but also by integrating into the existing systems for room planning and student data. (Flex, ElevPlan). We would not like to have a new system where we have to enter everything again. Without integration, there is no realistic scenario for usage.

- The room heating should be driven partly by the heat that is entering the windows.

- The school have 20 sky lights where 5 is smoke-escape hatches in case of a fire. These 5 can be controlled centrally, the rest can't. There are also a hallway with a glass roof on 1st floor but the windows on that is only manually operated. This rule is needed (If temperature on hallway is above 25 then open the windows. Close the windows in case of rain.)

- General equipment, especially projectors, are sometimes left on by students and teachers. They should be turned off based on week/time and room planning.

- The existing control system (CTS) for ventilation is hard to understand. The GUI is made like an electrical diagram, and is not user friendly. Moreover, the temperatures shown in the program is from sensors in the ventilation ducts, not in the room themselves, making it very hard to regulate them, since you'll have to change the settings and then wait for 10-15 minutes. Then you'll have to go to the room and manually hold a thermometer before going back and repeat this.

3. <u>Chief Consultant @ Bygningsstyrelsen</u>

Varme + ventilation med CTS (central tilstand styring)
    Sneider Honywell-produkter
    Gas, fjernvarme, osv.
    Ift. udetemperaturen., en kurve - i cts. "Når det er 0 sende man 45 grader, når"

Ventilation
    Nogle steder efter rum-temperatur. Nogle steder.

Varme
    Fjernvarme, primært
    Få med damp.

Helligdags-program
    Mest kontorejendomme - lav blus i weekender, ferier og heligdage.
    Sænker temp stopper ventilation. Også døgn basis i CTS. 7-17 ventilation.

Drifter statens kontorejendomme + universiteter (har egne driftsaftdelering.) + politi, domstole.

Udevendige persienner, solafskærmning (ift. en vejrstation) styrer hvornår de går ned nogle gange integreret i cts, eller standalone.

Drifter stærkstrøm, varme osv.

EBS - FN Byen Marmorkaj - der sidder følere (motion sensors), integreret i netværker - kan styre lys og solafskærmning og ventilation.

    Motion sensors mest til lys på gange, kontorer osv.
    Akkustiske sensorer - de bruger det en del. Efter kl. 9 ikke har været aktivit i en time, så slukkes.
        Lys på trappeopgange.

FN Byen - Marmorkaj

Problem med ældre bygning
    for dyrt at installere
    for dyrt at integrerer (så skal det være trådløse systemer)
    wifi er ikke god nok endnu, for mange ting der forstyrre radiator termostater.

Lukkede CTS systemer! bundet af leverandør

Der er systemer på markedet med åbne standarder ()

Skabe konkurerrence
    Service delene
        Programmering + Installerer et nyt ventilationsanlæg integreres med CTS.

CTS kræver uddannelse (minimum en halv time.)
    CO følere

4. Administrative Officer @ UNI-C

Existing Policies

- Light motion - Hørt andre der sige at vifte med arme og ben er irriterende. Måske efter et vist tidspunkt, idag er det hele dagen det virker sådan.
- Blinds hvis det stormer; Fint nok - sikring for dem. Ikke til anden gavn.
- Printerne går i sleep mode, og skal vækkes før man f.eks. kan scanne.

New Policies

- Gå automatisk ned når solen står lige ind på vinduet - skal kunne overrules. Det har noget med lysstyrken at gøre og ikke nødvendigvis temperaturen, da der bliver alt for lyst til at kunne arbejde. Går op igen når solen er normal igen.
- Når en medarbejder kommer ind i bygningen, kunne det være rart at computeren starter, så man ikke skal vente.
- Ville gerne optimerer så computerne slukkede automatisk for at spare energi, men der er mange problemer forbundet med det. Nogle medarbejdere har brug for at computerne kører hele tiden, pga. små arbejdsservere osv, andre har brug for at kunne tilgå den via VPN/fjernadgang, og hun kan ikke overskue om det kan lade sig gøre teknisk. Alle programmer skal være åbne med indhold. Må godt tænde og slukke hver gang man kommer/går.
- Kan den tætteste printer vækkes af sleep mode, når en medarbejder kommer ind om morgenen?
- Forskellige adgangskoder.
- Måske lys på kontoret når man bruger sit adgangskort?
- Vil have en cafe latte på borden når man ankommer om morgenen.
- Automatisk toilet-bræt-nedslå-mekanisme når man forlader toilettet.
- Hver morgen/nat skal opvaskemaskinen starte, hvis den er fyldt (enten vægt eller sensor). Problematik fordi folk ikke tømmer den, men stiller beskidt ind igen, hvilket så gør at den næste vasker det hele én gang til.
- Automatisk stille telefonen videre, når man forlader bygningen eller forlader sin plads. Men skal kunne virke med ens arbejdstider, så den ikke stiller videre når man har fri, men kun til f.eks møder osv. Det skal være muligt at angive forskellige numre den skal vidrestille til, evt. baseret på tid man er væk, eller om man forlader bygningen.