TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Merlin Kasesalu 164014IAPB

RULE-BASED CONTROL OF THE SPACE HABITAT UNDER THE CONDITIONS OF INCOMPLETE INFORMATION

Bachelor's thesis

Supervisor: Jüri Vain
Professor

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Merlin Kasesalu 164014IAPB

KUU HABITAADI ENERGIAKASUTUSE REEGLIPÕHINE JUHTIMINE MITTETÄIELIKU INFORMATSIOONI TINGIMUSTES

Bakalaureusetöö

Juhendaja: Jüri Vain
Professor

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Merlin Kasesalu

06.05.2020

Abstract

The present thesis is inspired by an international student project IGLUNA - a space habitat. The goal of the IGLUNA project is to design a smart building prototype for long-term space missions to Moon and Mars where vital living conditions must be maintained under extremal outdoor environment.

In this thesis, the rule-based control approach of heating and lighting the habitat is studied and a software prototype called Smart Adaptable Habitat (SAH from now on) is developed. SAH is a software that determines the intensity of light and the amount of power necessary at each moment to achieve or maintain the desired temperature conditions in the habitat. The required power for illumination is disregarded in the current software due to its negligible impact in comparison to the power needed for heating. The main purpose of the software lies in comfort and the efficient use of energy. The software takes into account the energy available in the habitat, the inhabitants' past preferences and their activities when predicting the desired temperature of every room, allowing for a more flexible and optimized timing and power settings than would be possible with manual control. Furthermore, it is required that the software can operate even in cases where some of the input data gets corrupt or is missing.

For software design and implementation, the principles of constraint logic programming and SWI-Prolog programming environment have been used. The result is tested with the 3D simulation created by Britta Pung.

This thesis is written in English and is 50 pages long, including 6 chapters, 22 figures and 4 tables.

Annotatsioon

Kuu habitaadi energiakasutuse reeglipõhine juhtimine mittetäieliku informatsiooni tingimustes

Antud lõputöö on inspireeritud rahvusvahelisest tudengiprojektist IGLUNA - kuu habitaat. IGLUNA projekti eesmärgiks on disainida nutimaja prototüüp pikaajalisteks missioonideks Kuule ja Marsile, kus elamistingimusi tuleb säilitada ekstreemses väliskeskkonnas.

Selles töös on uuritud reeglipõhist lähenemist habitaadi kütmisele ja valgustamisele ning see on realiseeritud prototüüpse tarkvarana. Antud tarkvara määrab valguse tugevuse ja vajamineva võimsuse, et saavutada ja hoida soovitud temperatuuri habitaadis. Valgustuse jaoks vajaminev võimsus on välja jäetud, sest selle kogus on võrreldes küttele kuluvaga minimaalne. Tarkvara põhieesmärk on mugavus ning efektiivne energiakasutus. Tarkvara võtab arvesse saadaval olevat energiat, elanike varasemaid eelistusi ja nende tegevusi, et ennustada elanike eelistatud temperatuuri igas toas. Ette ennustamine peaks lubama tarkvaral valida kõige optimaalsema ajastuse ning võimsuse soovitud temperatuuride saavutamiseks. Lisaks peab antud tarkvara toimima ka juhul, kui osadel väliskomponentidel tekib lühiajaline rike ning nende pakutavad sisendid on vigased või puuduvad.

Tarkvara disainimisel ning implementeerimisel on kasutatud loogilise programmeerimise põhimõtteid ja SWI-Prolog programmeerimiskeskkonda. Tulemust testitakse, kasutades Britta Pungi loodud 3D simulatsiooni.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 50 leheküljel, 6 peatükki, 22 joonist, 4 tabelit.

List of abbreviations and terms

DPI	*Dots per inch*
TUT	Tallinn University of Technology
SAH	Smart Adaptable Habitat
SN	Semantic Network
LP	Logic programming
CLP	Constraint logic programming
mgu	Most general unifier

Table of contents

1 Introduction 11
1.1 Project background 11
1.2 Main requirements and design assumptions 12
1.2.1 Energy consumption constraints 12
1.2.2 Habitat resident comfort constraints 13
1.2.3 Control adaptability to personal preferences 13
1.2.4 Flexibility and extendibility of the control software 13
1.3 Problem statement 13
1.4 Thesis overview and main results 14
2 Preliminaries 15
2.1 The principles of choosing system prototyping platform: imperative versus declarative programming 15
2.2 Logic programming 18
2.3 Constraint logic programming 21
2.4 CLP for reasoning with partial information 24
2.4.1 Unification using polymorphic inheritance 24
2.4.2 Unification using semantic distance (similarity of concepts) 27
2.4.3 Reasoning with partial unification 29
2.5 Decision validation using visual simulation 32
3 Control system design considerations 33
3.1 Habitat climate control 33
3.2 Control system architecture 33
3.3 Deciding the temperature control setpoints 35
3.4 Smart temperature control under energy constraints 36
3.5 Deciding the setpoints of light control 37
4 Implementation of the rule-based control system 38
4.1 Main function 38
4.2 Knowledge base 40
4.3 Outliers 41

4.4 Predicting temperature 41
4.5 Choosing the temperature based on the available energy 47
4.6 User changing the temperature 49
4.7 Choosing the light intensity 50
4.8 User changing the light 51
5 Validation of the implementation by visual simulation 52
5.1 Brief overview of the simulation solution 52
5.2 Communication between the simulation and controller 52
5.3 Demo scenario 54
6 Conclusion 61
6.1 Future developments 61
References 62
Appendix 1 – Index of rules 63
Appendix 2 - Repository link 65

List of figures

Figure 1. a) First order semantic net b) Relations semantic net 25
Figure 2. Rule for interpreting a binary relation..... 26
Figure 3. Predicate instance_of/2 is implemented using tail recursive rule 26
Figure 4. Example knowledge base..... 26
Figure 5. Search rule akin/4 that uses backtracking 28
Figure 6. Knowledge base describing the condition of a car..... 29
Figure 7. Diagnostic rules..... 29
Figure 8. Query to diagnose the condition of battery 29
Figure 9. Example of harness literals 30
Figure 10. Battery/2 query..... 31
Figure 11. Meta-rule of the form 31
Figure 12. Firm knowledge has the likelihood 1 31
Figure 13. The generic form of a CLP cause with n literals to be unified weakly..... 32
Figure 14. Conceptual architecture of the habitat control system 34
Figure 15. The temperatures are kept within the expected range..... 55
Figure 16. The light intensity depends on the room and the person in that room 56
Figure 17. Differences in bedroom temperatures 57
Figure 18. Anna is training in the gym and the temperature is around 20 degrees, matching the prediction. 57
Figure 19. Before manually setting the temperature for the room Relax (aka the living room) 58
Figure 20. After manually setting the temperature at 30 degrees for the room Relax ... 58
Figure 21. Changing the available energy to 0 makes all the power inputs go to 0 as well 59
Figure 22. Insufficient available energy 60

List of tables

Table 1. Simulation input files 52
Table 2. Simulation output files 53
Table 3. Light preferences for testing..... 55
Table 4. Preferences saved for testing the temperature 56

1 Introduction

1.1 Project background

The present thesis is inspired by an international student project IGLUNA - a space habitat [1]. The goal of the IGLUNA project is to design a smart building prototype for long-term space missions to Moon and Mars where vital living conditions have to be maintained in extremal outdoor environment. The prerogative of requirements to such buildings is the inhabitants' safety and comfort for the successful accomplishment of long-term missions. The functionality, reliability and resilience of the habitat's life-critical services depends primarily on its energy availability and efficient energy use. Moreover, long-term autonomous living sets also extreme standards to the requirements on lighting modes, air conditioning, temperature and all other inner climate parameters of the habitat that may have an influence on people's psychological and physical comfort.

In this thesis, specifically, the rule-based control approach of heating and lighting the habitat is studied and a software prototype called Smart Adaptable Habitat (SAH from now on) is developed. SAH is a software that determines the intensity of light and the amount of power necessary at each moment to achieve or maintain the desired temperature conditions in the habitat. The required power for illumination is disregarded in the current software due to its negligible impact in comparison to the power needed for heating. The main purpose of the software lies in comfort and the efficient use of energy.

By predicting the desired temperature of every room, the software should be able to choose the most efficient timing and power settings to achieve the trade-off between the efficient use of energy and comfort. Thus, the first goal of SAH is to make the power usage more economic than when manually operated. This could be critical in extreme conditions when the power supply is limited.

The temperature prediction also adds the factor of comfort, since the inhabitants do not have to plan the heating of each room on their own. Instead, the room will be heated to the temperature they wish for by the time they need it.

It means that all IGLU inhabitants' comfort targeted solutions must be considered within the context of safety and reliability concerns in the presence of severe conditions. For control rules to be applied in SAH it presumes that the control system does not fail when some of the control input data delay or get corrupted due to some system component or their communication failure. The most unreliable components are typically sensors that operate under hostile environmental conditions (cosmic radiation, temperature extremes).

In this thesis, the detailed taxonomy of habitat potential failures is not presented since it remains out of the thesis scope. Instead, it is assumed that some of the input data used for control decisions can be corrupted or temporarily missing. Thus, the focus of this thesis is on the design of the rule system that is applicable for control decision making without a drastic drop in the quality of decisions in situations where some of the decision data are missing or not available. To achieve this goal, it is assumed that the rule-based control to be developed in the thesis can ensure graceful degradation of the control system services till the recovery of the failing module or normal mode restoration by external means.

1.2 Main requirements and design assumptions

In the following section, the main requirements for the control system will be described that have to be taken into account when designing and programming the rule system.

1.2.1 Energy consumption constraints

The lowest required amount of energy per day is 16 kWh to keep the rooms at the lowest allowed temperature, which by default is 14 degrees. The default setting of distribution between the rooms is based on their size. The setting is modifiable manually but could be automated in future developments.

1.2.2 Habitat resident comfort constraints

For every activity there are comfort ranges, which can be divided into zones - “ideal”, “good”, “difficult”, and “unsuitable” (i.e. the activity is not possible or recommended in such conditions).

1.2.3 Control adaptability to personal preferences

The control system tries to provide optimal conditions for activities by taking into account the inhabitants’ preferences. Higher priority activities are given a greater weight during optimization while taking into account that the person should be able to carry out the function in a suitable environment. Although the priorities are modifiable, the default setting gives the highest priority to the activities based on how difficult it would be for the person to conduct their activity in a different room.

1.2.4 Flexibility and extendibility of the control software

The decision-making rules need to change along with changing circumstances - setpoints, criteria weights, the rules may adopt additional conditions (Horn clauses make take on additional literals or alternatives when details are added to the rules).

1.3 Problem statement

The goal of the thesis is to develop rule-based control software for controlling the internal climate (temperature and lighting) of the space habitat IGLUNA. For software design and implementation, the principles of constraint logic programming and SWI-Prolog programming environment are recommended for use. The decision rules implemented in control software must take into account personal preferences of the residents of the habitat and obey the situation-dependent energy consumption constraints. The software must be reliable and resilient to failures of components that are supplying the control rules with the necessary input data. Furthermore, the software should maintain the capability of providing feasible decisions in the presence of corrupted or missing input data.

1.4 Thesis overview and main results

This thesis offers an overview of the programming tools used and their advantages over the alternatives, a description of the algorithms for determining the temperature and light, and the key rules used to implement said algorithms. The result is a validated controller system for managing a habitat's temperature and lighting based on the inhabitants' preferences and the available energy.

2 Preliminaries

2.1 The principles of choosing system prototyping platform: imperative versus declarative programming

When choosing the implementation platform for IGLUNA control software two main options are considered:

a) using some standard programming language such as Java, C++, Python,
b) applying constraint logic programming (CLP) that support declarative programming style.

The basis of imperative programming is describing the steps needed to be taken in order to change the state of the computer [2]. Declarative programming, however, defines what the desired state is, instead of how it should be achieved [3]. While imperative programming focuses on how things should be done, declarative focuses on what needs to be done without the how.

The criteria being followed when deciding on the relevant programming framework are following [4]:

- Possibility to represent functionality on high level of abstraction;
- Complexity of coding;
- Automatic problem-solving support;
- Easy software maintenance.

In the rest of the Section the programming alternatives are compared with respect to the given criteria.

Possibility to represent functionality on a high level of abstraction. CLP supports declarative modelling by constraints, i.e. the description of properties and relationships between partially known objects. It also allows correct handling of precise and imprecise, finite and infinite, partial and full information.

Complexity of coding. One of the main advantages of declarative programming is the smaller amount of code compared to imperative, which comes from not having to define

the step-by-step procedures. This also makes it easier to manage the code. The programming process consists of two phases: specifying the problem as a set of constraints and solving it where solving is automatic.

Another important feature of CLP is the flexibility of programming, since constraints can easily be added, removed or modified.

In imperative programming the length of code grows rapidly with the complexity of a problem, making it difficult to navigate in the program [5].

Focusing on the definitions of the desired states while keeping the instructions on how to achieve them abstract also makes adding new features as well as optimization simpler [5].

Automatic problem-solving support. CLP solvers solve combinatorial problems efficiently. It is easy to combine the constraint solving with search and optimization strategies. Another advantage is the presence of built-ins: constraint solvers offer numerous built-in strategies and algorithms for solving constraint models, e.g. propagation of the effects of new information (as constraints).

Easy software maintenance

- Modularity. Keeps predicates and operations local and allows independent development of software in different modules.
- Standard interface with other programming languages such as Java, C++, Python, allows embedding modules written in these languages in Prolog code and embedding Prolog modules in programs written in these languages.
- Graphical tool support for program tracing and debugging enables stepwise execution of Prolog programs and visualization of data and program call stack.
- Injection of hooks and spy points in the code for easy reaching deeply nested program states.
- Visual threads monitor enables the monitoring of threads and their memory consumption running in parallel
- Pretty printing of clauses, predicate dependencies and terms

All these features help in debugging, maintenance, and updating the code that is important for fast prototyping of system functionalities and compilation of the whole software release.

Conclusion. Under the given selection criteria CLP programming paradigm has several advantages over imperative languages. In particular, CLP provides a generic framework for

- modeling with partial information and with infinite information,
- reasoning with new information,
- solving combinatorial problems.

Hence, CLP will be used as the implementation platform for IGLUNA rule-based control system.

2.2 Logic programming

Logic programming (LP) is a programming paradigm that is based on formal logic. The principles of LP are implemented in languages such as Prolog, Answer Set Programming (ASP) and Datalog. The most popular of them, Prolog, has been developed in various versions both for academic and commercial use. LP is used for modelling, creating executable specifications, solving combinatorial problems such as scheduling, planning, timetabling but also for analysis, simulation, verification, and diagnosis of software, hardware and industrial processes.

The applications of LP in research include program analysis, robotics, agents, protein folding, genomic sequencing, linguistic parsing and many other domains where expert knowledge needs to be encoded and incorporated in decision support systems. One of the most prominent examples of Prolog uses is the natural language processing engine in IBM Watson artificial intelligence [6]. The reason why Prolog is considered so powerful in AI is because it enables for easy management of recursive methods, and pattern matching. The recursive rules allow efficient representation of reachability in parse trees and the operation of negation-as-failure to check the absence of conditions.

A logic program consists of a set of logical expressions - facts and rules about some problem domain. All these logic expressions are written in the form of *Horn clauses* or *Horn rules*. Horn clause can be understood as logic implication in the form

$$H :- B_1, \ldots, B_n. \qquad (1)$$

where H is called the *head* of the rule and comma separated literals B_1, ..., B_n are called the *body*. Facts are unconditional rules that have no body, and they are written in the simplified form:

$$H.$$

An alternative but logically equivalent to formula (1) is Horn clause representation in disjunctive form where H is the only positive literal and $\neg B_1, \ldots, \neg B_n$ are negative literals

$$H \vee \neg B_1 \vee \ldots \vee \neg B_n.$$

A literal is an atomic formula or its negation. An atomic formula of form $P(t_1,..., t_n)$ represents a logic predicate of arity n where P is the predicate symbol or predicate functor and its arguments $t_1, ..., t_n$ are terms which do not include any other nested predicate symbols.

Intuitively, the Horn clause has to be understood as follows: the assertion `H` is true if all the assertions $B_1, \ldots, B_n$ of the rule body are true. The assertions $B_1, \ldots, B_n$ may be simple atomic formulae or literals referring to the heads of other rules. Logic programs also have a procedural interpretation as goal-reduction procedure: to solve `H`, solve B_1, and ... and solve B_n.

The recursive reduction procedure of the program is implemented by means of two inference rules *resolution* and *factorization*, and term transformation procedure called *unification*. The unification of terms is needed to apply the resolution and reduction rules on Horn clauses that differ only by terms occurring in clauses arguments. The unification substitutes variables with terms. Let $\{ x_1, \ldots , x_n \}$ be a set of variables. The substitution of variables is denoted by $\{x_1/t_1, \ldots , x_n/t_n\}$, where x_i denotes the variable to be substituted and t_i – the substituting term. It is required that t_i differs from x_i.

<u>*Definition*</u> (*unification of terms*):

Terms t_1 and t_2 are unifiable if and only if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$.

As a rule, the unification results in renaming or concretization of terms. In LP, there exists the *minimal substitution principle* according to which the unification has as few substitutions as possible. This is to not restrict the substitutions for future unifications. Therefore, for making the inference rules applicable only the necessary set of terms are unified. The *minimal substitution principle* leads to the notion of *most general unifier* (*mgu*).

<u>*Definition*</u> (*most general unifier*):

The most general unifier of terms t_1 and t_2 is the substitution ρ that satisfies following conditions:

- ρ is unifier of terms t_1 and t_2;
- for any other unifier σ of terms t_1 and t_2 there exists a unifier τ such that $\sigma = \tau \circ \rho$, where $\circ$ denotes the composition of unifiers.

From definition it follows that for any term t following equation holds:

$\sigma(t) = \tau(\rho(t))$.

It has been shown independently by Martelli, Montanari (1976) [7] and Paterson, Wegman (1978) [8] that there exists linear-time algorithm for computing *mgu* for any pair of terms.

The practical implication of this result is that before applying LP inference rules, the literals that have same functor and arity in different clauses are tried to be unified. Thus, the resolution and factorization rules that presume such unification are formulated in (*) and (**) respectively.

Resolution rule

$$\frac{\neg \boldsymbol{A_1} \vee A_2 \vee \ldots \vee A_n \qquad \boldsymbol{B_1} \vee B_2 \vee \ldots \vee B_m}{(A_2 \vee \ldots \vee A_n \vee B_2 \vee \ldots \vee B_m)\sigma} \qquad \sigma = \mathrm{mgu}(\boldsymbol{A_1}, B_1) \qquad (*)$$

Factorization rule

$$\frac{\boldsymbol{A_1}, \boldsymbol{B_1}, A_2, \ldots, A_n}{(\boldsymbol{A_1}, A_2, \ldots, A_n)\,\sigma} \qquad \sigma = \mathrm{mgu}(\boldsymbol{A_1}, \boldsymbol{B_1}) \qquad (**)$$

Though the early versions of Prolog were designed to support only 1st order logic reasoning by using unification and inference rules (*) and (**), the later versions were extended to support also a resolution with higher order terms. For instance, SWI-Prolog version 8.0.3 has term constructor "`=../2`" and `system` predicates that allow constructing and invoking terms dynamically during program execution. Such system predicates are `call/2` and `apply/2` which enable higher order reasoning using the SWI-Prolog inference engine. The rules that employ higher order logic constructs are elaborated in Section 2.4 as the theoretical contribution of the thesis.

2.3 Constraint logic programming

Constraint logic programming (CLP) is a form of constraint programming, in which logic programming is extended with the concepts from constraint satisfaction theory. CLP program is a logic program that contains constraints in the body of clauses. Let formula (2.1)

$$\underline{G}(\bar{X}) :- F(\overline{Y}) \qquad (2.1)$$

denote a clause where $F(\bar{X})$ is a conjunction of constraints and literals the parameters of which are variables of vector $\bar{X}$. The constraints may have very different form starting from simple arithmetic constraints, e.g. X>Y and ending with complex recurrent equations. The clause in (2.1) states that the statement *G* holds under the constraints in *F* and while all literal *s* in *F* are true.

Like in regular logic programming, the query specifies a proof goal that triggers inference to prove the validity of that goal. In CLP the query may contain constraints in addition to literals. A proof for a goal is composed of clauses whose bodies are satisfiable constraints and literals that can be proved using other clauses.

The execution of the goal is performed by an interpreter, which starts from the goal and recursively scans the clauses trying to prove the goal. The constraints encountered during this scan are placed in a set called the *constraint store*. If this set is found out to be unsatisfiable, the CLP interpreter backtracks, trying to use other clauses for proving the goal. In practice, satisfiability of the constraint store may be checked using an incomplete algorithm, which does not always detect inconsistency. [9][10]

In addition to the regular unification of variables that is like in LP, CLP also supports variable quantification, conditional answers and the easy symbolic manipulation of formulas.

In addition to resolution and factorization, automatic constraint reasoning includes two methods: constraint propagation and simplification. Thus, CLP extends LP in two aspects:

1) derivations in LP can be expressed as CLP derivations;
2) CLP answer is a set of constraints while LP answer is substitution. [9][10]

Constraint programming consists of three basic techniques:

- Declare the domains of the variables;
- Declare the constraints on the declared variables;
- Search for the solutions.

Constraint Programming Process:

- Constraint Modeling: Representations of a problem as a constraint satisfaction problem with constraints is called constraint modeling.
- Constraint Solving: Solving the constraint models formulated by modeling can be carried out using any of the three methods:
    - Domain specific method (Simplexe, Gröbner bases etc);
    - General Method (Constraint propagation);
    - Combination of both methods (Solver cooperation).

Constraints in Constraint Programming are relationships between variables or unknowns, each taking a value in a domain. They restrict possible values that a variable can take. Constraints need to be identified.

Properties of constraints:

- Constraints may specify partial information, i.e. they do not necessarily specify the values of variables.
- Constraints may be heterogeneous, i.e. they specify relations between variables with different domains.
- Constraints are declarative, they specify what variable relationships may hold.
- Constraints are additive, i.e. the conjunction of constraints is important.
- Constraints are rarely independent, i.e. they share variables.

Constraints in CLP are typically classified as follows:

- Equality and inequality constraints
- Boolean Constraints
- Linear Constraints
- Arithmetic Constraints
    - Integer intervals and finite integer domains

- Extended intervals of real
- The set of reals

Constraint solving methods can be divided into domain specific and general methods.

1. Domain specific methods are:

   - special purpose algorithms devoted to specific domains and constraints;
   - method by means of specialized packages called constraint solvers.

   The examples of domain specific solving methods include:

   - Programs that solves systems of linear equations
   - Packages for linear programming
   - Implementations of unification algorithm

   In CLP practice, the domain specific methods are preferred over general methods, should they be available.

2. General constraint solving methods are adopted to different types of constraints and domains (variables). They are targeted at reducing the search space with specific search methods. There are two different method groups:

   - Constraint Propagation Algorithms. These are the algorithm that repeatedly remove inconsistent values from the domains and reduce the search space. They also maintain equivalence while simplifying the problem and achieve various forms of local consistency of constraints.
   - Search methods explore the search space. They consist of a combination of constraint propagation, backtrack, branch and bound search. [9][10]

2.4 CLP for reasoning with partial information

In this Subsection the extension to regular LP and CLP rules are defined that employ higher order logic operators and enable reasoning with partial information and use weaker forms of term unification.

2.4.1 Unification using polymorphic inheritance

For the representation and processing of application knowledge we use the notations of semantic network (SN) theory. The facts that describe the status of the world, e.g. the habitat and inhabitants' condition are usually specific instances of more general concepts. Layering the concepts and relations between them by their level of abstraction helps in minimizing the form of knowledge representation that is needed for logic inference and control decision making.

The knowledge organized as semantic network is composed of concepts and relations between these concepts. Both concepts and relations can be of a different level of abstraction. Schematically, the ground level facts and the concepts they are derived from (by applying inheritance) are depicted in Figure 1. As such, each concept may have its own set of attributes, plus those inherited from its ancestor concepts. The inheritance relation named "`is_a`" connects a concept with its immediate ancestor concept. Because of polymorphism the concepts in a semantic network may have several ancestors, and instead of a pure tree structure we get a grid structure.

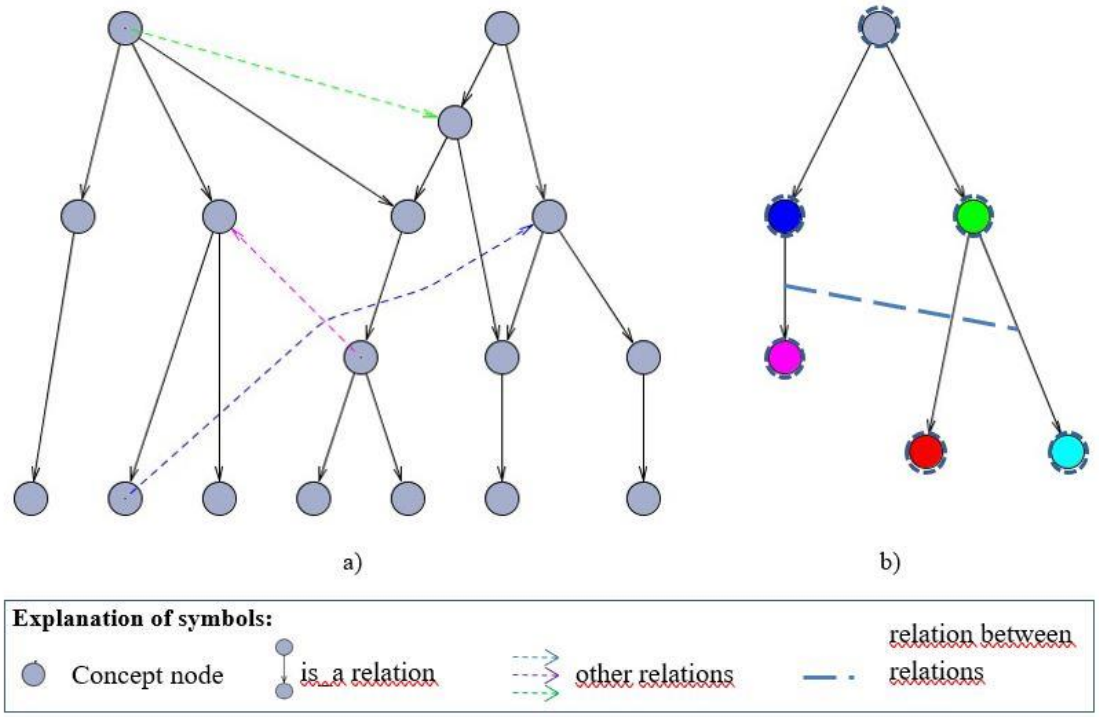

Figure 1. a) First order semantic net b) Relations semantic net

Nodes (denoted with a dotted line) of the semantic network in Figure 1.b) encode the relations of Figure 1.a). The colors of lines and nodes identify the same relations. The bold line in Figure 1.b) denotes a second order relation, i.e. the relation between the first order relations shown in Figure 1.a)

Based on this recurrent construct we can define knowledge structures with high order relations (possibly with many level of abstraction).

For a practical use of a recurrent SN-s we also need inference rules to reason on recurrent structures.

We can generalize the example to all binary relations as shown in Figure 2. Here, the rule name for interpreting a binary relation is `metapredicate` where its parameters are `Predicate` bound with the predicate name, `Conc1` and `Conc2` are the parameters bound with the names of the concepts in this relation. The literal `instance_of/2` in the body of the rule has two parameters: the first one bound with the argument concept of the rule head and the second one `Conc2` should be unified with the first argument `Conc1` or its ancestor in this concept's `is_a` hierarchy. This rule succeeds if both concept arguments unify with some concepts, which are the argument values themselves

or their ancestors and if they are in the relation specified with the argument `Predicate` or its ancestor in the SN.

```
metapredicate(Predicate, Conc1, Conc2):-
     (Predicate=P ; instance_of(Predicate,P)),
     instance_of(Conc1, AConc1),
     instance_of(Conc2, AConc2),
     Term =.. [P, AConc1, AConc2],
     Term.
```

Figure 2. Rule for interpreting a binary relation

```
instance_of(X,Y):- is_a(X,Y).
instance_of(X,Y):- is_a(W,Y), instance_of (X,W).
instance_of(X,X).
```

Figure 3. Predicate instance_of/2 is implemented using tail recursive rule

For a better comprehension of the working of the explained rules we demonstrante it with a simple example.

Example:

```
is_a(talk, communicate).
is_a(debate, talk).
is_a(quarrel, debate).
is_a(peter, uncle).
is_a(anni, aunt).
```

Figure 4. Example knowledge base

Assume the facts in Figure 4, which specify the `is_a` relation and one binary relation `communicate/2` between concepts `uncle` and `aunt`

```
communicate(uncle, aunt).
```

Then the query

```
?- metapredicate(debate, peter, aunt).          (*)
```

returns true because the concepts `peter` and `aunt` have the ancestor concepts `uncle` and `aunt` which are in relation `communicate` which in turn is the ancestor of the relation `debate` given as argument value in the query (*).

2.4.2 Unification using semantic distance (similarity of concepts)

An alternative way of defining the weak unification of terms is to use their semantic distance in the SN. The distance measure or *similarity of concepts* in SN can be evaluated on a scale 0 - 100. This provides the possibility to define the "similarity threshold" that is needed for pruning the search tree and deciding whether the concepts in the relation unify (weakly) with some other concept that is “similar enough” to replace them in case there is no exact match in the knowledge base. We call it the *weak unification* relation between the clauses that encode concepts.

The similarity predicate tries to unify the sets X and Y of parameter names of clauses A and B to be unified and returns the percent of equivalent parameter names (of concepts in the SN) and the result of their unification. Having two terms $A(X)$ and $B(Y)$ as arguments, the similarity predicate has the general form $similar(A(X), B(Y), P, mgu(A(X \cap Y), B(X \cap Y)))$ where *mgu* denotes the *most general unifier* and the similarity P is calculated by the formula

$$P = \frac{|mgu(A(X \cap Y), B(X \cap Y))|}{|X \cup Y|} * 100$$

Two semantic units in SN have a similarity value 100 if all their parameters have the same identifiers and they unify. The value is 0 if none of them have an identical name or the ones with the identical names do not unify. Here the system predicate **`same_term`**`(@T1, @T2)` is used for deciding on the identity of the concepts’ argument terms.

To search in SN for similar concepts the `instance_of/2` predicate (defined in Section 2.4.1) should be relaxed. For that, instead of finding only ancestor nodes in SN all those nodes in SN are searched which are in the same transitive closure (of either relations `is_a` or its inverse is_a^{-1}) with $A(X)$, i.e. this search covers a set of $B(Y)$:

$$\{B(Y)\ .\ \ A(Y) \in is\_a^{+} \cup (is\_a^{-1})^{+} \Rightarrow\ B(Y) \in is\_a^{+} \cup (is\_a^{-1})^{+}\ \},$$

where is_a^{+} and $(is\_a^{-1})^{+}$ denote a transitive closure of the relations is_a and is_a^{-1} respectively.

Transitive closure has a connection in SN with the argument concept via is_a relation, i.e. including also other child nodes of common ancestor nodes. So, instead of instance_of/2 we use in the metapredicate/3 a new search rule akin/4 that uses backtracking for finding the best match *B* to concept *A*.

```
akin(A,_,_,_):-
      assert(best_known(_,0,[])),
      transitive_closure(is_a),
      invert(is_a, is_a_1),
      transitive_closure(is_a_1),
      (closure(A,C,_) #==> closure(B,C,_);
      closure(C, A,_) #==> closure(C,B,_)),
      similar(A,B,P, Unified_parameters),
      update_best(P,B, Unified_parameters),
      fail.

akin(A,B,Score,Unified_parameters):-
      retract(best_known(B,Score, Unified_parameters)).

update_best(Score,B, Unified_parameters):-
      best_known(BB, Best_score,_),
      Score > Best_score,
      retract(best_known(_,_,_)),
      assert(best_known(B,Score, Unified_parameters)),!.
      update_best(_,_, _).

transitive_closure(Rel):-  % finding 1st power of the relation
      Relation =..[Rel,X,Y],
      call(Relation),
      assert(closure(1,X,Y)),
      fail.

transitive_closure(_):-    % finding i+1st power of the relation
      call(closure(I,A,B)),
      call(closure(1,B,C)),
      I1 is I+1,
      assertz(closure(I1,A,C)),
      fail.

transitive_closure(_).
```

Figure 5. Search rule akin/4 that uses backtracking

2.4.3 Reasoning with partial unification

An alternative way (compared to the one discussed in in 2.4.2) of computing weak unification is to count how many literals in the bodies of unifiable clauses are unifiable. The need for such unification can be motivated as follows.

The knowledge base that includes facts about runtime sampling values from sensors may easily be incomplete because of erroneous sensor readings, missing rules or delayed updates. This can easily block the logic inference for control decision making. We illustrate this case with a simple example.

Example: Assume there are expert rules for diagnosing the cause of car malfunction.

Suppose the facts in Figure 8 describing a car's conditions are stored in the knowledge base.

```
lights(dim).
co2(over_norm).
windglass(with_crack).
engine(hard_start).
```

Figure 6. Knowledge base describing the condition of a car

The diagnostic rules that allow estimating the need for repair have the form depicted in Figure 7.

```
battery(full):-
      engine(quick_start),
      lights(bright).
battery(empty):-
      engine(does_not_start),
      lights(dim).
```

Figure 7. Diagnostic rules

```
?- battery(X).
false
```

Figure 8. Query to diagnose the condition of battery

As it can be seen, the query returns false because the fact `engine(does_not_start)` is missing from the knowledge base and unification of the literal `engine(does_not_start)` in the body of the second clause fails, although it is likely that the battery may be empty even by one criteria. Hence, although there is expert knowledge stored in the knowledge base the inference returns almost no information about it.

To avoid such situations, we add so called *harness literals* to the rule body that allows applying resolution even in case some of the literals in the rule body do not have a unifying clause in the knowledge base. These *harness literals* have parameters that characterize the likelihood of the derived conclusion. The example in Figure 9 illustrates the use of harness literals. For brevity only the second clause of the battery rule is decorated with harness literals (highlighted in red).

```
% Main rule
battery(empty, Likelihood):-
      reset,
      engine(does_not_start), incr_valid ;true),
      incr_all_possible,
      (lights(dim)), incr_valid ; true),
      incr_all_possible,
      valid(Available), all_possible(AllPossible),
      Likelihood is Available/AllPossible.

% Assisting predicates

      incr_valid:-
            retract(valid(Current)),
            NewCurrent is Current + 1,
            assert(valid(NewCurrent)).

      reset:-
            retractall(valid(_)),retractall(all_possible(_)),
            assert(valid(0)), assert(all_possible(0)).

      incr_all_possible:-
            retract(all_possible(Current)),
            NewCurrent is Current + 1,
            assert(all_possible(NewCurrent)).
```

Figure 9. Example of harness literals

After harnessing the rules, the extended query

```
?- battery(Condition, Likelihood)
```

returns a more informative result saying that with likelihood 0.5 the car battery is empty, as shown in Figure 10.

```
?- battery(Condition, Likelihood).
      Condition = empty
      Likelihood = 0.5
      true
```

Figure 10. Battery/2 query

```
rule(…,P):-
      (rule1(…, P1), PList1 = [P1]; true), C1=1,
      (rule2(…, P2), PList2 = [P2|PList2]; true), C2 is C1+1,
      …
      (ruleN(…, PN), PListN=[PN|PListN-1]; true), CN is CN1+1,
      P is listsum(PListN, Sum)/CN, !.
```

Figure 11. Meta-rule of the form

To generalize the construct we get the meta-rule of the form in Figure 10, where rule is the functor of the original clause head, rule1 to ruleN are functors of literals 1 to N in the original clause body, … denotes the parameters in the head and in the body of literals, P1…PN denotes the variables that are unified with the likelihood values of body literals. The rule returns in parameter P the weighted arithmetic mean of N literals' likelihoods

$$P = \frac{\sum_1^N P_i}{N}.$$

In case the literal refers to some Prolog fact the likelihood value must be explicitly defined, e.g. a firm knowledge is specified with likelihood value 1.

```
fact(…, 1).      % fact that represents valid knowledge
```

Figure 12. Firm knowledge has the likelihood 1

Note that the way of extending normal rules with harness literals is uniform and can be integrated with any Prolog parser.

```
clause_head(…):-
      wu(literal_1(X_1),XX_1), constraint(XX_1),
      …
      wu(literal_n(X_n), XX_n), constraint(XX_n).
```

Figure 13. The generic form of a CLP cause with n literals to be unified weakly

To conclude, a CLP clause with *n* literals to be unified weakly is used in combination with constraints on weakly unified terms, and has the generic form of Figure 13, where `wu/2` denotes a weak unification operator (discussed in subsections 2.4.1 - 2.4.3), `X_i` denotes the parameters list of `literal_i`, and `XX_i` denotes the valuation of weakly unified parameters of `X_i`. The predicate `constraint/1` can be any Prolog constraint that is a correct type with respect to `XX_i`.

2.5 Decision validation using visual simulation

For testing the software, we are using a simulation created by Britta Pung, which allows us to monitor the changes of temperature and light in real time.

The goal of validation is to confirm that the software:

1. Works according to the inhabitants' preferences;
2. Can manage with an amount of energy that is less than what is necessary for providing ideal conditions.

The goals can be tested by specifying the inhabitants' preferences and the amount of energy available, and by monitoring the changing of the temperature and light in the simulation and analysing the extent to which the results match the expectations.

3 Control system design considerations

3.1 Habitat climate control

The relationship between temperature, time and power (which correlates with energy usage) is derived from the formula (3.1)

$$\frac{\Delta T}{t} = \frac{P}{c \times \rho \times V}\ , \qquad (3.1)$$

where

ΔT	is the change in temperature in the room in Celsius,
t	is the time of heating in seconds,
P	is the heater's power in Watt,
ρ	is air density in the room in kg / m^3,
V	is volume of the room in m^3,
c	is the specific heat capacity of air, which is 1006 J / (kg × C).

The formula (3.1) is derived as follows.

$Q = c \times \Delta T \times m$ (3.2) [11]
$P \times t = c \times m \times \Delta T$ (3.3) [12]
$P \times t = c \times V \times \rho \times \Delta T$

One of the limitations appears in the case where the desired temperature is lower than the current temperature. In such case the power would be negative, which would imply the need for an air conditioner. Spending energy on an air conditioner in an environment in which heat is scarce is not justified. Therefore, cooling down is achieved by decreasing the heater's power supply to 0. Hence, there is no direct way to regulate the time needed to reach a colder temperature in the room. This is regulated indirectly by the parameters of ventilation.

3.2 Control system architecture

The functionality of the habitat's climate control system can be divided into three groups: sensing, control and actuation. By the level of control abstraction two layers are presented: direct control and supervisor control. The direct control layer includes the feedback control loops that keep the physical characteristics of the environment at some setpoint value that is computed on the supervisor control level. For direct control, the physical sensor signals of temperature, light, sound, humidity are filtered, processed and

sent to the controller, which calculates the input signals for the actuators. The actuators transform the physical characteristics of the environment according to their input. The direct control layer actuation module includes heaters, air filters, light sources with modifiable intensity and speakers for sound background. The last is for influencing the inhabitants' emotional mood. The general architecture with two control layers is depicted in Figure 14.

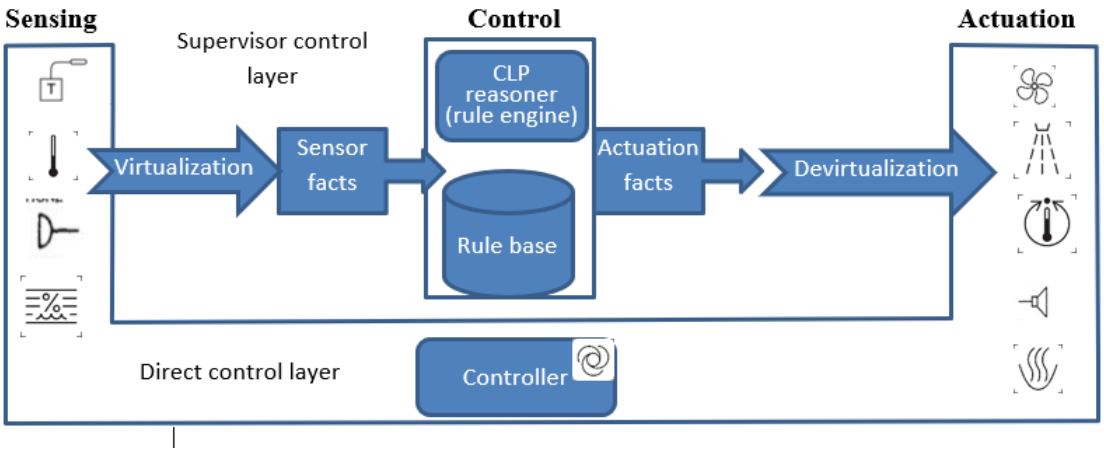

Figure 14. Conceptual architecture of the habitat control system

The direct control level sensors include:

1) Temperature sensors for measuring the temperature in each room;
2) Positioning sensors for detecting the room where an inhabitant resides in at any given moment. This could possibly be a combination of a face detector camera and a movement detector between the rooms;
3) An activity sensor for categorizing the activity of each inhabitant at any given moment. This would have to be a combination of a camera and a movement recognition algorithm, and/or bodily sensors to recognize the activities. The categories could be as follows: training, working, eating, relaxing, sleeping;
4) An electricity meter for measuring the power consumption at different segments of the habitat.

The controller takes sensor samplings and calculates the amount of energy available. Based on the available energy resource the control modes are decided. This is combined with an algorithm that divides the energy between the control loops. heaters and other devices that need to be powered.

The physical actuators powered within different control loops are heaters, lights, ventilators, and recreation utilities (music, TV, air cleaners, etc).

The supervisor control layer involves virtual twins of sensing and actuation devices that are needed for high-level control, e.g. calculating new setpoints for direct control loops, learning the consumption profiles from the inhabitants' routine monitoring and activity plans. The virtual twins also maintain the data about the components of the direct control layer, e.g. their functioning mode, reliability status ageing, usage profiles and other.

The current thesis focuses on the design and implementation of the supervisor level software using the principles of CLP and SWI-Prolog programming environment. In the following subsections the concrete control loops are discussed and the supervisor level control principles outlined.

3.3 Deciding the temperature control setpoints

The ideal temperature prediction is based on the inhabitants' preferences. The preferences are recorded whenever an inhabitant changes the room temperature settings. The preferred temperature is saved along with a timestamp, the room, every person currently in that room and every person's current activity. This makes it possible to associate certain preferences with a timeframe, an activity and a room for every person. The underlying assumption here is that even though only one person changes the temperature, they have consulted previously with the other people in the room to make sure that they are all right with the new temperature.

The controller predicts the temperature at a modifiable interval; the default is 15 minutes. To predict the ideal temperature for a room in 15 minutes, the controller begins by finding the likeliest people to be in that room and their likeliest activity in 15 minutes. It then finds everyone's likeliest preferred temperature, trying to find preferences that match with the timeframe, room and activity. If there is no such data in the database, the controller tries to find matches only with the room and activity. If there are no such matches either, it removes the activity requirement as well.

If there are more than one person predicted to be in the room, the controller calculates the average of their preferences, taking into account their predicted activity. The preferences of the inhabitants predicted to be training will be given weight by the factor of 3, the ones working will be given by the factor of 2. The levels of priority are based on how difficult it would be for the person to conduct their activity in a different room. Since there is only one room with training equipment available, it would be the most difficult for those who are training to relocate. It might be easier to relocate when you are working and it should be the easiest to relocate for eating, relaxing or sleeping.

Next, the required preparation time and power will be calculated and the temperature preparation is initiated.

3.4 Smart temperature control under energy constraints

For simplification the available energy is a constant that shows how much energy can be spent on temperature in an hour, that can be manipulated to test the software's performance in different situations. In reality, there would need to be a sensor that detects the amount of energy in storage and an algorithm that calculates how much of the energy can be spared for temperature and how much for other devices.

The available energy is divided between the rooms based on priority. The default setting is for the energy set aside for a given room to be proportionate to the size of the room. This can be modified in case there are rooms that by default require a higher or a lower temperature, for example labs or the bedroom. If there is a room that needs more energy for its lowest allowed temperature than is set aside for it, all rooms are set to their lowest temperatures to ensure that no room falls below the allowed limit.

The procedure begins by calculating the required energy for the desired temperature in 15 minutes. If the required energy exceeds the available energy, the interval is doubled, i.e. if the interval is 15 minutes and the desired temperature cannot be achieved in that time within the energy constraints, the controller calculates how much energy would be needed to reach the ideal temperature in 30 minutes. If the energy constraints are still not met, the controller starts lowering the desired temperature by 0.2 degrees until the energy restrictions are met or the lowest allowed temperature is reached.

3.5 Deciding the setpoints of light control

Light is turned on when the first person enters a room and turned off when the last person has left the room. Once a person has entered the room, the lighting depends on the activity of the people in the room. The light preference of the person conducting the activity of the highest priority is chosen. The order of priorities is as follows: training, working, eating, relaxing, sleeping, following the logic outlined in 3.3. If people have the same activities but different preferences, the average is chosen.

4 Implementation of the rule-based control system

In the following chapter, the main rules and facts are outlined and explained. To make them easier to find in the source code, an index has been added to the annex, which specifies the module in which each of the rules can be found.

4.1 Main function

The control loop is periodically initiated by querying the controller with the rule `start/0`.

The periodic process that repeats every second is achieved with the predicate `repeat/0`, which provides an infinite number of choice points [13], and `get_time/1`, which returns the current time as a timestamp [14]. “A TimeStamp is a floating point number expressing the time in seconds since the Epoch at 1970-01-01” [15]. The timestamp is written in a file and the following timestamps are then compared against it, while both of the timestamps have been rounded up to 0 decimal places. Once the new timestamp is bigger than the one in the file, the file is overwritten with the new timestamp and the predicates succeed, continuing to the literals after them.

Every second the following predicates are called:

- `update_current_activity/0`,
- `update_current_temperature/0`,
- `add_user_temperature/0`,
- `add_user_light/0`,
- `update_available_energy/0`,

which check for pertinent inputs from the simulation.

The rule `update_current_activity/0` has a literal `add_activity/1`, which checks whether an inhabitant has moved on to a new activity and if they have, the previous activity is added to the knowledge base with the beginning and end times.

Then `change_light/1` (see Subsection 4.7) is applied for every inhabitant.

Next, `set_power_fact/1` is applied for every room. It first checks for the dynamic fact `new_user_temperature/2` to see, if a user has modified the temperature anywhere. If they have, the protocol outlined in Subsection 4.6 is followed.

If `new_user_temperature/2` fails, the dynamic fact `power_calculation_time(Room, Calculation_timestamp)` is called, which specifies the time at which a new temperature should be calculated for this room. If the timestamp is equal to or smaller than the current timestamp, the new temperature is calculated by the protocol outlined in Subsections 4.4 and 4.5.

To signal the time at which a new power command needs to be given to the simulation, a new dynamic fact

```
power(Room, Power, Power_command_timestamp),
```

where

Power_command_timestamp is the current timestamp,

is asserted.

Then, to set the time for the next power calculation, the interval is converted to the simulation's time and then added to the current timestamp. The resulting timestamp is used to assert the dynamic fact `power_calculation_time/2`, which specifies the next time the power needs to be calculated.

Even though, given the static interval, it would be easier to create the power calculation with a similar design to the `start/0` predicate, i.e. the power is calculated after the timestamp has increased a certain amount, the current design with dynamic facts gives more flexibility for future developments by allowing to vary the times of changing the power and for calculating a new power setting. Ideally, the controller should be able to account for the durations of certain activities and modify the intervals for calculating the power accordingly.

4.2 Knowledge base

The activities of the inhabitants are stored in the module dynamic_data in the form of the following fact template:

```
activity(Person, Activity, Room, Beginning_hour,
Beginning_minute, End_hour, End_minute),
```

where

Person	is the inhabitant conducting the activity,
Activity	is the activity being conducted,
Room	is the room in which the activity is being done,
Beginning_hour	is the hour of the start time,
Beginning_minute	is the minute of the start time,
End_hour	is the hour of the end time,
End_minute	is the minute of the end time.

For simplification, both the simulation and the controller only have five activity categories: training, eating, sleeping, relaxing and working.

The individual temperature preferences are stored in the module dynamic_data in the form of the following fact template:

```
temperature(Temperature, Person, Room, Activity, Year, Month,
Day, Hour, Min).
```

The preferred light intensity is stored in the form of the following fact:

```
light(Light, Person, Room, Activity, Year, Month, Day, Hour,
Min).
```

The facts used to reflect the current situation are the following:

- `current_temperature(Room, Temperature, Year, Month, Day, Hour, Minute),`
- `available_energy(Energy),`
- `current_activity(Person, Room, Activity, Year, Month, Day, Hour, Minute).`

4.3 Outliers

When predicting the preferred temperature and light, the average of various preferences is calculated. On account of sensor failures or other exceptional occurrences, it is possible for some of the preferences to vary greatly from the rest, which could have an unwanted effect on the average. To prevent this, before the calculation of an average, the possible outliers are always removed from the set of sensor samplings. The rules for calculating the outliers are in the math module.

For finding the outliers, the interquartile range method is used, which counts values below the lower fence and above the upper fence as outliers.

The lower fence is found with the following formula

$Q1 - 1.5 \times IQ$

and the upper fence is found with

$Q3 + 1.5 \times IQ$

where

Q1 is the lower quartile, which is the 0.25×Nth (or 0.25×(N + 1)th) value, i.e. the median of the lower half,

Q3 is the upper quartile, which is the 0.75×Nth (or 0.75×(N + 1)th) value, i.e. the median of the upper half,

IQ is the interquartile range, i.e. Q3 - Q1. [16][17]

4.4 Predicting temperature

The temperature is predicted in the temperature module with the rule

`decide_temperature(Room, Current_H, Current_M, Minutes, Temperature),` (4.1)

where

Current_H is the hour of the current time,

Current_M is the minute of the current time,

Minutes is the time interval to the goal point in time,

Temperature is the decided temperature.

Rule (4.1) is composed of

- `min_from_current(Current_H, Current_M, Minutes, H, M)` (4.2),
- `likeliest_people_in_the_room(Room, H, M, People)` (4.3),
- `all_prefs_average(People, Room, H, M, Temperature)` (4.4).

If there are no recorded preferences that match the criteria, the default temperature for the room is chosen.

The time for which the temperature is needed is calculated with

`min_from_current(Current_H, Current_M, Minutes, H, M)` (4.2).

The likeliest people to be in the room are predicted with

`likeliest_people_in_the_room(Room, H, M, People)` (4.3),

where

Room is the room for which this rule applied,

H is the hour of the goal,

M is the minute of the goal,

People is the list of people predicted to be in room Room at H:M.

The average of the preferences of the people who are predicted to be in the room at the goal time is calculated with

`all_prefs_average(People, Room, H, M, Temperature)` (4.4)

where

People is the list of people whose preferences are taken into account,

Room is the room for which the temperature is calculated,

H is the hour of the time,

M is the minute of the time,

Temperature is the found average.

The rule `likeliest_people_in_the_room(Room, H, M, People)` (4.3) uses the rule

`likeliest_room(Person, H, M, Likeliest_room)` (4.5)

where

Person	is the inhabitant,
H	is the hour,
M	is the minute,
Likeliest_room	is the likeliest room where Person will be at H:M.

The likeliest room is found by selecting a random room from a list of all the rooms Person has been in at H:M. A single room is added to the list of rooms for each time it has been recorded in the database that Person has been in it at H:M. Therefore, the room in which the person has been the most often at the given time is statistically the likeliest to be chosen by a random selection.

The rule `all_prefs_average/5` (4.4) finds the preferences with the rules

- `likeliest_activity(Person, H, M, Room, Activity)` (4.6),
- `likeliest_preferred_temperature(Activity, Room, Person, H, M, Temperature)` (4.7).

It (4.4) queries the priority of the activity, which each person will be doing, to determine how many times the preference should be added to the list of preferences from which the average will be taken.

The likeliest activity of each person is found with the rule in the energy module

`likeliest_activity(Person, H, M, Room, Activity)` (4.6),

where

Person	is the inhabitant,
H	is the hour of the time,
M	is the minute of the time,
Room	is the room in which the activity will take place,

Activity is the found activity.

The rule (4.6) tries to unify with

```
every_activity(Person, Activity, Room, H, M, Duration),
```

which gives as a result every Activity the Person has been doing in the Room at H:M.

If it fails, (4.6) tries to unify with

```
every_activity(Person, Activity, _, H, M, Duration),
```

excluding the argument Room.

As outlined in 4.2, the instances of activities are stored in the database as facts with their beginning and end time (validity period). Whether or not an activity has taken place at a certain time is determined with the rule

```
during(Beginning_hour, Beginning_minute, End_hour, End_minute,
Current_hour, Current_minute),
```

which uses the literal

```
earlierOrSame(Beginning_hour, Beginning_minute, End_hour,
End_minute),
```

which is true, if the time at Beginning_hour:Beginning_time is earlier or the same as the time at End_hour:End_minute. For `during/6` to be true, the current time needs to be later than or equal than the beginning time of the activity and earlier or equal than the end time of the activity.

The complication involved here is that time is a cycle, not linear, so there is no clear-cut predetermined point, which can be considered the ending and beginning of the cycle.

One predetermined point is midnight, but the problem arises when recording the time of sleep. If the restart point is midnight, then `earlierOrSame/4` would have to be true, if the Beginning_hour is smaller than the End_hour.

If a fact about a person sleeping has been recorded from 22:00 to 7:00, then it would be impossible to find a time that would satisfy `during/6`, since every hour that is bigger than 22 is also bigger than 7. Therefore, additional rules need to be added.

The proposed rules are:

1. End_hour is smaller than or equal to 12 and Beginning_hour is smaller than End_hour;
2. Beginning_hour is bigger than or equal to 12 and smaller than End_hour;
3. Beginning_hour is bigger than 19 and End_hour is smaller than 12;
4. Beginning_hour is equal to End_hour and Beginning_minute is smaller than or equal to End_minute.

The rules 1. - 3. apply to activities between:

1. 0:00 - 12:59;
2. 12:00 - 23:59;
3. 20:00 - 11:59.

These rules exclude activities that start before midday and end after midday, and activities that start before 20:00 and end after midnight. The assumption for the former is that people change activities more frequently during midday, e.g. working or training and eating during lunch time or relaxing, and therefore the chance of missing an activity is small and can be considered trivial. The assumption for the latter exclusion is that people engage in shorter activities in the evenings and / or go to sleep before midnight.

Both exclusions could be avoided by adding an additional rule to recording the inhabitants' activities that has a breakpoint at 12:00 and 20:00. Hence, if a person starts an activity before 12:00 or 20:00 and continues them after the breakpoints, then two entries are saved for the activities - one that ends at 11:59 / 19:59 and one that starts at 12:00 / 20:00. However, this would also require additional rules when calculating the duration of activities.

The likeliest preferred temperature is calculated in the temperature module with the rule

`likeliest_preferred_temperature(Activity, Room, Person, H, M, Temperature)` (4.7),

where

Activity	is the activity the person is predicted to be doing,
Room	is the room in which the person is predicted to reside,
Person	is the person whose likeliest preferred temperature is calculated,
H	is the hour of the time at which the prediction is supposed to happen,
M	is the minute of the time at which the prediction is supposed to happen,
Temperature	is the likeliest preferred temperature.

The first choice is based on time preference, which uses the following rule:

`every_temp_preference_likeliest(Activity, Room, Person, Current_H, Current_M, Temperature)` (4.8),

where

Activity	is the predicted activity,
Room	is the room for which the temperature is being predicted,
Person	is the person's whose preference is being predicted,
Current_H	is the hour of the current time,
Current_M	is the minute of the current time,
Temperature	is the likeliest preferred temperature.

The rule finds all the temperature/9 facts (see Subsection 4.2) that unify with

`temp_preference_time_likeliest(Activity, Room, Current_H, Current_M, H, M)` (4.9),

where

H	is the hour of temperature/9,
M	is the minute of temperature/9.

The rule (4.9) finds if the recorded preference was recorded in a timeframe that is relevant, considering the current time. To do so, it first calculates the average of the durations of the recorded instances of the activity, excluding the outliers. Secondly, it

calculates the interval from the current time to the time of recording of the temperature preference and the interval from the time of recording to the current time. If either of those intervals is less than or equal to the average duration of the activity, the preference is counted relevant.

If rule (4.8) fails, meaning there are no preferences that fit the timeframe in the database, rule (4.7) tries to unify with `temperature/9` (see Subsection 4.2), taking into account the person, room and activity. If it fails as well, it discards the activity and only takes into account the person and the room.

Once (4.7) has found a list of temperature preferences, the outliers are removed from the list and the average is calculated, which will be given as the person's likeliest preferred temperature.

4.5 Choosing the temperature based on the available energy

The rule

`update_available_energy/0`

is executed every minute to check whether the available energy has been changed and if it has, then it updates the dynamic fact `available_energy/1`.

The required power supply is chosen with the rule

`preparation_time_desired_temp(End_temp,Room,Desired_time_in_min, Change_time,Power)` (4.10),

where

End_temp	is the temperature setpoint (C),
Room	is the room in which the temperature is being changed,
Desired_time_in_min	is the time it would ideally take to achieve the temperature (min),
Change_time	is the time it will actually take to achieve the temperature (min),

Power is the amount of power that will be given to the heater (W).

Firstly, it calls

`is_there_enough_energy_for_min_setting_in_all_rooms/0` (4.11)

in the energy module, which checks whether there is enough available energy to provide enough power for all the rooms to be at their lowest allowed temperature. If (4.11) fails, then every room is given a percentage of the available energy according to their priority.

If (4.11) does not fail, the current temperature and the desired temperature are compared. If the desired temperature is lower than the current temperature, the power is set to 0 W to allow the room to cool down on its own.

If the desired temperature is equal to or higher than the current temperature, the required energy is calculated with

`required_energy(Start_temp, End_temp, Room, Change_time, Power, Req_energy),`

where

Start_temp is the current temperature (C),

End_temp is the desired temperature (C),

Room is the room in which the temperature is being changed,

Change_time is the time it will take to change the temperature (s),

Power is the amount of power that will be needed (W),

Req_energy is the required amount of energy (J).

It will calculate the required energy and power to achieve the temperature and maintain it for an hour. The required energy to achieve the temperature is calculated with the literal `change_temp/6`, which uses the formula

$E = c \times \Delta T \times \rho \times V$,

where

E is the required energy,

c is the specific heat capacity of air,

ΔT is the change in temperature,

ρ is the density of air,

V is the volume of the room.

The energy required to maintain the temperature is calculated with the literal `keep_temp/3`, which has two literals: `temp_loss/2` and `change_temp/6`. `Temp_loss/2` gives the temperature of the room after 1 minute with no heating. For simplification it is assumed that the temperature drops by 0.1 degrees every minute without heating. `Change_temp/6` is then used to calculate the amount of energy needed to make up for the loss in temperature.

If the required power is less than or equal to the maximum limit of power (2000 W by default) and the required energy is less than or equal to the energy available for that room, the predicate succeeds.

In case of failure, the desired_time_in_min (a parameter of 4.10) is lengthened by a minute each retraction until the previously mentioned constraints are met or the desired_time_in_min becomes longer than twice the length of the predetermined interval.

If the desired_time_in_min has been lengthened to twice the length of the interval, the setpoint temperature begins to be decreased by 0.2 degrees at a time until the constraints are met or the lowest allowed temperature is reached.

4.6 User changing the temperature

The rule

`add_user_temperature/0`

checks every second whether the user has modified the temperature in the simulation. If they have, a new dynamic fact `user_temperature/7` is asserted and `add_all_temp_preference/2` adds the chosen temperature as a preference for everyone in that room at that given time.

Then `preparation_time_desired_temp/5` (4.10) calculates the required time and power to achieve the user's preferred temperature or the closest possible temperature. The controller starts preparing the room for the user's desired temperature right away.

At the end, the dynamic fact is retracted.

4.7 Choosing the light intensity

The light is controlled with the predicate `change_light(Person)`. Controlling the light is based on the movements of every inhabitant and the predicate is only used when a person's location changes.

If a person goes from one room to another, the light in the previous room is turned off, if the room was left empty.

The light for the room the person enters is decided with

```
decide_light(Room, Light).
```

The rule finds the people currently in the room and takes into account their preferences based on their activities and the priorities of the activities (see Subsection 3.3).

Then the intersection of the people who are currently in the room, in the middle of an activity with the highest priority, and who have relevant preferences is found.

The rule

```
relevant_preference(Room, Activity, Person)
```

tries to unify with the facts

```
current_activity(Person, Room, Activity, _, _, _, _, _)
```

and

```
light(_, Person, Room, Activity, _, _, _, _, _).
```

By doing so it tries to find the people in the room that are doing the activity with the highest priority and also have a recorded light preference that matched that room and activity.

If `light/9` fails, the room argument is removed and only the preferences that match with the person and the activity are looked for. If this also fails, only the preferences that match with the person and the room are looked for.

Once all the people with the relevant preferences have been found, their preferences are found (either those that match with the room and activity, only the activity or only the room) and the average of those is calculated.

4.8 User changing the light

The rule

```
add_user_light/0
```

checks every second whether the user has modified the light in the simulation. If they have, then a new dynamic fact `user_light/7` is asserted. That light is set as the current light for the room and the light setting is added as a preference from everyone in the room. At the end, the dynamic fact is retracted.

5 Validation of the implementation by visual simulation

5.1 Brief overview of the simulation solution

The validation of the software is conducted with the 3D simulation built by Britta Pung with the Unreal Engine 4. The purpose of the simulation is the visualization of the controller behaviour to validate the algorithmic efficacy and the functional correctness. The simulation is a 3D model of the moon habitat, which is divided into nine separate rooms - 3 bedrooms, 2 laboratories, a gym, a training room, a bathroom, a storage room and a living room. The simulation has a numeric display of the temperature and light intensity in each room, and the different levels of the light intensity are visualized.

The simulation also shows three inhabitants, who move around and engage in activities, namely working, relaxing, eating, training and sleeping. The three inhabitants are named after the three members of the IGLUNA group - Britta (in grey), Merlin (in pink) and Anna (in green). The user can change the temperature and lighting in each room, which is then stored as a preference for the inhabitants in that room. This allows the simulation of the daily life of the inhabitants in the moon habitat.

The available energy is also modifiable by the user to help validate the system's behaviour in cases of energy scarcity, which could be effected by various environmental factors or the malfunction of the energy production systems.

The time in the simulation is sped up to 4 minutes in 1 second.

5.2 Communication between the simulation and controller

The communication between the simulation and controller is conducted through text files. The files are divided between purposes and each file is only for one-way communication, i.e. it receives input either from the simulation or the controller.

Table 1. Simulation input files

File name	Description	Contents	Example of contents
tasks_light.txt	Task from the controller to change the light intensity in a room	room, light intensity, timestamp	bed1,20,2019.05.09-17.0000
tasks_power.txt	Task from the controller to change	room, power in Watts, timestamp	relax,0.5,2019.05.09-19.00.00

	the power of a heater in a room		

Table 2. Simulation output files

File name	Description	Contents	Example of contents
user_energy.txt	Used when the user changes the available energy constant in the simulation.	Energy, year, month, day, hour, minute.	available_energy_kwh(2500, 2019, 5, 13, 5, 0).
user_light.txt	Used when the user changes the light intensity in a room.	Room, light intensity, year, month, day, hour, minute.	user_light(relax, 10, 2019, 5, 11, 10, 56).
user_temperature.txt	Used when the user changes the temperature in a room.	Room, temperature, year, month, day, hour, minute.	user_temperature(relax, 23.0, 2019, 5, 13, 5, 16).
current_temperatures.txt	Gives the current temperatures of every room.	Room, temperature, year, month, day, hour, minute.	current_temperature(bed3, 20.0, 2019, 5, 13, 5, 0).
current_lights.txt	Gives the current light intensity of every room.	Room, light intensity, year, month, day, hour, minute.	current_light(training, 100, 2019, 5, 13, 5, 0).
current_activities.txt	Gives the current location and activity of every inhabitant.	Inhabitant, room, activity, year, month, day, hour, minute.	current_activity(merlin, bed2, sleeping, 2019, 5, 13, 7, 0).

Tasks for the simulation are written with the predicate

```
write_task(Filepath, Task),
```

where

Filepath is the path of the file,

Task is the task for the simulation.

Before writing the task,

`file_is_empty(Filepath)`

checks whether the file is empty, i.e. the last task has been read and deleted by the simulation. If `file_is_empty/1` fails, `time_guard/1` is used, which waits for a specified number of seconds before checking again whether the file is empty. The specified amounts are 1 second, 1 second, 2 seconds and 10 seconds. If the file is still not empty, the task is ignored.

When reading the files user_temperature, user_light or user_energy, `file_is_empty/1` and `file_is_marked_empty/1` are used to check whether there is a new user input in the file. They fail if the file is not empty and the content is not 'empty', which means new input has been added. After reading it, the file is overwritten with the text 'empty' to let the simulation know that the file is ready for new inputs. 'Empty' is written instead of an empty string, since in SWI-Prolog the written text must be in the form of a term.

When accessing a file, `catch/3` is used to prevent errors from disrupting the system. A common disruption was caused by the error "no permission to open source_sink". The error appeared to not follow a detectable pattern - it was caused by different files at different times that the controller had no trouble accessing the rest of the time. The possibility of a coding error related to an unclosed stream was eliminated. The current hypothesis is that the error is caused by a synchronization error where both the simulation and controller try to access the same file at the same time. On account of the infrequency of the error, the decision was made to skip reading the file in case of the error. Further work is needed to eliminate the error or to ensure its triviality in relation to its effect on the controller.

5.3 Demo scenario

In order to monitor the behaviour of the controller, it needs to be run together with the simulation. The data about each of the rooms in the upper right corner shows the temperature and light intensity in each room. During runtime, the simulation (Figure 15) shows that the temperatures in each room are kept within the expected range. The lighting is turned on and off in rooms, depending on whether someone is in that room or not. By defining the inhabitants' preferences, it is possible to validate the correctness of the light intensity. To test whether the controller incorporates the inhabitants'

preferences when calculating the temperatures, specific test preferences are predetermined in the knowledge base.

Figure 15. The temperatures are kept within the expected range

To test the lighting, the preferences in Table 3 were saved for Merlin (in pink), Anna (in green), Britta (in grey).

Table 3. Light preferences for testing

Person	**Room**	**Activity**	**Light intensity**
Merlin	Living room	Relaxing	100
Merlin	Living room	Relaxing	60
Merlin	Living room	Relaxing	40
Anna	Bedroom 3	Sleeping	100
Britta	Bedroom 1	Sleeping	0

Figure 16 shows Merlin in the living room relaxing and the light intensity at 66, which is the average of her previous preferences. Anna is sleeping in bedroom 3 and the light intensity is at 100, which matches the set preference. Britta is working in lab 2 and the light intensity is at 80, which is the default light setting, since Britta does not have any recorded preferences with working or lab 2.

Figure 16. The light intensity depends on the room and the person in that room

The temperature validation process is complicated for two main reasons. Firstly, the temperatures are constantly changing and are the result of both the predictions of the controller as well as the attempts at achieving and maintaining a certain setpoint. Secondly, the temperatures are based on the inhabitants that are predicted to be in a room at a given time, not who is actually in the room, hence without an exhaustive knowledge base of the inhabitants' activities it is possible for the temperature to not match with the preferences of the people in the room.

To test the temperature, the preferences in Table 4 were saved.

Table 4. Preferences saved for testing the temperature

Person	**Room**	**Activity**	**Temperature**
Merlin	Bedroom 2	Sleeping	14
Britta	Bedroom 1	Training	30
Anna	Bedroom 3	Sleeping	20
Merlin	Living room	Relaxing	17
Britta	Living room	Relaxing	23
Anna	Living room	Relaxing	20
Merlin	Gym	Training	17
Anna	Gym	Training	20

Britta	Gym	Training	25

Figure 17 displays the differences in the temperatures of the bedrooms. Bedroom 1 is around 30 degrees as per Britta's preference, bedrooms 2 and 3 are lower in temperature and are cooling down (the power of the heaters is 0).

Figure 17. Differences in bedroom temperatures

Figure 18. Anna is training in the gym and the temperature is around 20 degrees, matching the prediction.

Figure 19. Before manually setting the temperature for the room Relax (aka the living room)

Figure 20. After manually setting the temperature at 30 degrees for the room Relax

Figures 19 and 20 show the effect of manually changing the temperature in a room. Even though the set temperature is 30 degrees, the temperature in the room climbs up to 37 degrees before the power is turned off. Since this only happens if the desired increase in temperature is big, meaning that the power input is also big, it can be assumed that

the issue is not with the behaviour of the controller, but with the delay of communication between the software and the simulation.

The simulation does not allow for decimal places in the user input. Typically, the needed energy in an hour stays under 1 kWh. In order to test the behaviour of the controller under the condition of scarce available energy, the conversion rate is changed in the software, so that 1 kWh in the software is equal to 0.01 kWh in the simulation.

Figure 21. Changing the available energy to 0 makes all the power inputs go to 0 as well

Figure 22. Insufficient available energy

Figure 22 shows the case of insufficient available energy, which can be confirmed with the logs. The available energy is considered insufficient, if there is at least one room for which its designated portion of the energy is not enough to maintain its lowest allowed temperature. In this case, every room is given only its portion of the available energy.

Logging allows to check for exceptions during the runtime, which disrupt the communication between the controller and the simulation. The exceptions are caught, and the system continues to run smoothly with no disruptions visible in the simulation.

6 Conclusion

The result of this thesis is a rule-based control software for controlling the internal climate (temperature and lighting) of the space habitat IGLUNA. The software is capable of predicting the inhabitants' preferences and giving appropriate instructions to achieve them whilst taking into account the given energy constraints. New preferences are added to the knowledge base every time an inhabitant changes the temperature or lighting in a room and each preference is saved with references to the room, the person, their activity and time. Future predictions try to match with as many relevant references as are available. The software can also operate even in cases where some of the input data gets corrupt or is missing.

The behaviour of the controller can be monitored and validated through the 3D simulation built by Britta Pung.

6.1 Future developments

The delivered software is a basic prototype for a smart house controller, which could serve as the foundation for a more elaborate AI with the addition of new features or the integration with other controllers. New features could include a dynamic energy division algorithm and an algorithm for setting the interval between power calculations depending on the predicted length of an activity. Furthermore, a feedback system that confirms or rejects predictions could improve the accuracy of the predictions. This software could be integrated with another that monitors the inhabitants' physical and mental well-being and makes temperature and lighting suggestions based on that.

References

[1] Swiss Space Center. IGLUNA 2019. [WWW] https://www.spacecenter.ch/igluna/#igluna2019 (08.05.2020)

[2] Imperative programming. [WWW] https://www.computerhope.com/jargon/i/imp-programming.htm (08.05.2020)

[3] Declarative programming. [WWW] https://www.computerhope.com/jargon/d/declarprog.htm (08.05.2020)

[4] Benefits of constraint programming. [WWW] https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_intro_benef_cp.html (08.05.2020)

[5] Imperative programming: Overview of the oldest programming paradigm. [WWW] https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/ (09.05.2020)

[6] Jones, M. T. The languages of AI. 2017. [WWW] https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-languages-artificial-intelligence/ (09.05.2020)

[7] Martelli, A., Montanari, U. Unification in linear time and space: A structured presentation. Istituto di Elaborazione della Informazione, Consiglio Nazionale delle Ricerche, 1976.

[8] Paterson, M. S., Wegman, M. N. Linear unification. — *Journal of Computer and System Sciences*, 1978, 16, 2, 158-167.

[9] Frühwirth, T., Abdennadher, S. Essentials of Constraint Programming. Series: Cognitive Technologies. Springer, 2003.

[10] Frühwirth, T. Constraint Handling Rules. Cambridge University Press, 2009.

[11] Specific Heat Formula. [WWW] https://www.softschools.com/formulas/physics/specific_heat_formula/61/ (16.05.2020)

[12] How long would you have to yell to heat a cup of coffee? [WWW] https://www.physicscentral.com/explore/poster-coffee.cfm (16.05.2020)

[13] Predicate repeat/0. [WWW] https://www.swi-prolog.org/pldoc/man?predicate=repeat/0 (16.05.2020)

[14] Predicate get_time/1. [WWW] https://www.swi-prolog.org/pldoc/doc_for?object=get_time/1 (16.05.2020)

[15] Time and date data structures. [WWW] https://www.swi-prolog.org/pldoc/man?section=dattimedata (16.05.2020)

[16] Montis, K., Peil, T. First Quartile and Third Quartile. 2010. [WWW] http://web.mnstate.edu/peil/MDEV102/U4/S36/S363.html (16.05.2020)

[17] What are outliers in the data? — *NIST/SEMATECH e-Handbook of Statistical Methods,* 2012. [WWW] https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm (16.05.2020)

Appendix 1 – Index of rules

Main module:

- add_activity/1
- add_all_temp_preference/2
- add_user_light/0
- add_user_temperature/0
- change_light/1
- set_power_fact/1
- start/0
- update_available_energy/0
- update_current_activity/0
- update_current_temperature/0,

Temperature module:

- all_prefs_average/5
- decide_temperature/5
- every_temp_preference_likeliest/6
- likeliest_people_in_the_room/4
- likeliest_preferred_temperature/6
- min_from_current/5
- preparation_time_desired_temp/5
- temp_preference_time_likeliest/6

Energy module:

- change_temp/6
- during/6
- earlierOrSame/4
- every_activity/6
- is_there_enough_energy_for_min_setting_in_all_rooms/0
- keep_temp/3
- likeliest_activity/5
- likeliest_room/4

- required_energy/6

Light module:

- decide_light/2
- relevant_preference/3

Appendix 2 - Repository link

https://gitlab.cs.ttu.ee/mekase/igluna