

UiO : **Department of Informatics**
University of Oslo

Semantic Modeling of Smart Homes in the ABS Real-time modeling language

Ming-Chang Lee and Olaf Owe
Research report 457, October 2016

ISBN 978-82-7368-422-6

ISSN 0806-3036



Semantic Modeling of Smart Homes in the ABS Real-time modeling language

Ming-Chang Lee and Olaf Owe

October 2016

Contents

1	Introduction	2
2	Smart Home Components	4
2.1	The interactions of each component in the Smart Home System .	5
2.2	Behavioral Specification	6
3	Modularized Interfaces	8
3.1	The Control Unit interface and subinterfaces	9
3.2	Modeling Smart Home System	9
3.3	The Home Owner interfaces	11
3.4	The Service Provider interfaces	11
3.5	The Grid Provider interfaces	12
3.6	The Distributed Database interfaces	12
3.7	Device interfaces	12
4	Implementation	14
5	Conclusion	15
5.1	Future Work	15

Chapter 1

Introduction

Smart homes together with smart grids and related smart technologies are becoming an essential part of our infrastructure. These technologies are expected to play an increasingly important part of our future lives, and in ways that are not yet known. New paradigms for the smart home will be developed, including useful and less useful ones.

Smart technology in the home is made possible with the presence of a smart meter in each home, allowing frequent energy readings, and with a gateway to connect other smart meters and the service providers through concentrators. Smart meters might generate a vast number of energy meter readings, which may be stored in a distributed database using cloud technology. Customer data contains private information with the possibility of misuse. The management of stored data and the associated cloud structure may have severe implication with respect to privacy and security.

In this paper we will build a formal framework for high level modeling of smart homes and smart technologies, focusing in particular on the interplay between home owners, smart devices, a control unit with smart meter, grid providers, and service providers. The interaction between these actors is interesting with respect to energy efficiency, system performance, the level of user control, privacy issues, security issues, and as a business model for energy related service providers. The modeling framework offers a number of abstractions that enable a high-level understanding of smart technologies, security issues, and privacy issues, and the communication of smart meters and smart devices. Our models are executable and formal, allowing prototyping, simulation, and qualitative analysis.

The purpose of the framework is to explore and experiment with possible smart home architectures and software solutions, and use simulation and formal analysis to evaluate aspects of their usefulness including security and privacy aspects. For instance a formal framework would make it possible to compare different smart home architectures. For us it would be interesting to compare three different smart home architectures:

- one where user data is stored under the control of each home owner (“fog” solution), resulting in a decentralized data management solution, and
- one where user data is stored by the service provider in a logically centralized manner (which may be physically implemented in a decentralized

fashion).

- one where user data is stored by each home owner and the service provider in a logically multi-tier manner (which may be physically implemented in a hierarchical fashion).

We wish to compare these three with respect to flexibility and efficiency of the grid provider, home owner, and service providers – and also with respect to privacy and security issues. In this work we focus on the second architecture, and report initial results, including experiments with that architecture.

For the modeling framework we use (an extended version of) the Abstract Behavioral Specification (ABS for short) modeling language [2], which is executable, object-oriented supporting concurrent objects, and allows formalization of architectures that can be reused in more low-level languages such as Java and C++. It has a formal semantics and is oriented towards specification of functional and non-functional properties. The ABS language builds on the concurrency model of Creol [1]. Relevant work on ABS has extended it to modeling of cloud architectures, with *deployment* control [4].

Even though the language allows detailed implementation design, we focus on abstract designs and high-level information flow. We do not here focus on implementation designs. For implementation, the Smart Home system can be implemented by Hadoop YARN and Spark Streaming can be used to process data flows of the whole Smart Home System shown in Fig. 1. In the future, we plan to compare implementation results and simulation results to check whether these two results match or not.¹

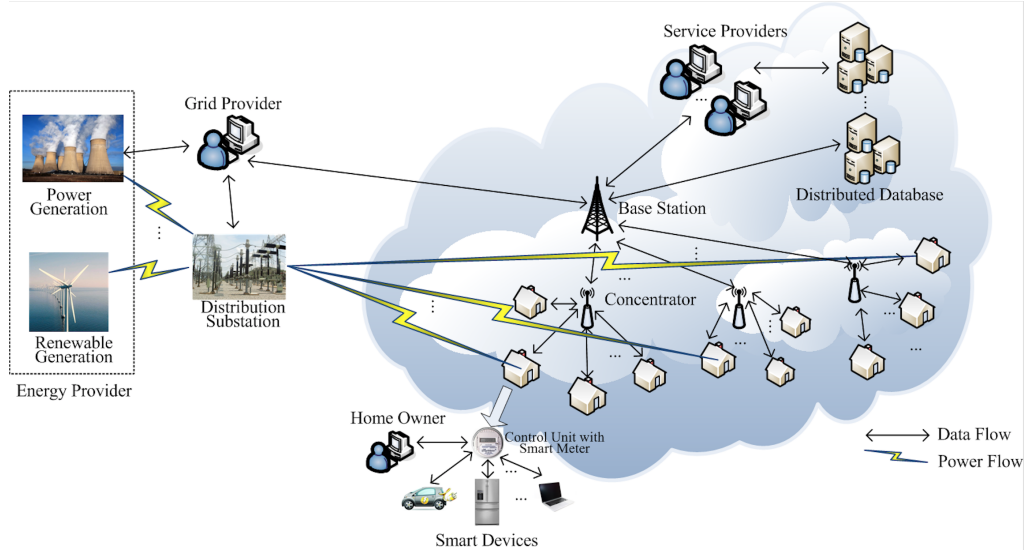


Figure 1.1: The architecture of Smart Home System.

¹The contribution of that work is that we can use the implementation result to prove the simulation result is correct, such that the simulation result can be more convincing. After that people can use this simulation model directly to set up different setting of Smart Home to check which one is better.

Chapter 2

Smart Home Components

Fig. 1 shows the architecture of Smart Home System. The components of the Smart Home System include:

1. Smart Meter: Each smart meter in the house will periodically collect data and transmit these data to Distributed Databases. Data may include energy used, energy created (by utilizing solar panels), Voltage, and Volt-Amps (kW/KVA) [Weaver2014]. When the data is transmitted to Distributed Databases, it will be encrypted to keep security and privacy against cyber attacks [Lichtensteiger2010].
2. Control Unit: There is a control unit, which can be used to control the smart devices of the smart home and the Smart Meter that is one of components of the Control Unit, as well as the interaction with the grid and service providers.
3. Smart Devices: There are a number of smart devices/appliances in a smart home. They can be controlled by the Home Owner directly or through the Control Unit.
4. Distributed Database: Distributed Database can be accessed by the Service Providers, a Grid Provider, and the Home Owner as shown in Fig. 1. It can be implemented by Hadoop Distributed File System (HDFS for short). HDFS provides a lot of benefits such as scalable, cost effective, flexible, fast, and resilient to failure.
5. Home Owner: The energy-consumption data consumed by Home Owner will be sent to Distributed Databases and stored in it. According to the privacy of Smart Home, the Home Owner shown in Fig. 1 can review or delete his/her data from the Distributed Database.
6. Service Provider: Each service provider shown in Fig. 1 provide different services for the Home Owner to subscribe. Different subscriptions provide different services, for instance controlling the energy usage in smart ways. These setting will be stored in the Control Unit and Distributed Database.
7. Energy Provider: the Energy provider in Fig.1 provide power generation or renewable generation to smart homes.

8. Grid Provider: It provides two services to a smart home: One is establishing the connection to the Energy Provider and the other is transportation of energy. It may also receive and redirect energy from a home producing energy, for instance solar energy. If the Home Owner's neighbor has solar energy, the neighbor also can be a Energy Provider with a separate local network. The Grid Provider will record what kind of energy that the Home Owner uses for his/her smart home in the Control Unit and Distributed Database.

The data flows of the Smart Home System include:

1. The energy-consumption data of a Smart Home is transmitted from a Control Unit to the Distributed Databases through intermediate devices, such as concentrators and a base station.
2. The Home Owner control smart devices by using the Control Unit, and reviews/accesses their personal data from the Distributed Databases through the Control Unit.
3. The Service providers and Grid Provider set up their services in the Control Unit, and review and analyse Home Owner data in the Distributed Databases.
4. The Grid provider communicates with Distribution Substations.

If the capacity of the Smart Grid system is not enough to process these data flows, the capacity of the system will be automatically increased to process these data flows. Otherwise, the capacity of the system will be automatically decreased.

The service provider can utilize the data in Distributed Database to do several big data analysis. Based on the results of analysis, service companies can make several corresponding decisions, e.g., increasing or decreasing energy provision based on different seasons automatically. On the other hand, the service provider also can detect malicious attacks or recognize anomalous behaviors by using the big data analysis. Once a smart home was attacked by an attack, the other smart homes at least will not be suffered by the same attack again.

2.1 The interactions of each component in the Smart Home System

Different services can be programmed and stored on the Control Unit, possible in interaction with the Home Owner, the Grid Provider, the Service Provider, the Smart Devices, and the Distributed Database as shown in Fig. 2.

The Control Unit can be programmed and set up by Home Owner, Service Providers, or Grid Provider to control smart devices, but the recording function of energy consumption for the house, i.e., the Smart Meter, will not be programmed by anyone. Also, before the Control Unit executes a mission that is submitted by Home Owner, Service Providers, or Grid Provider, Distributed Databases must make a record for this mission first.

The Control Unit provides several functions for Home Owner to choose to control the Smart Home through settings, e.g.,

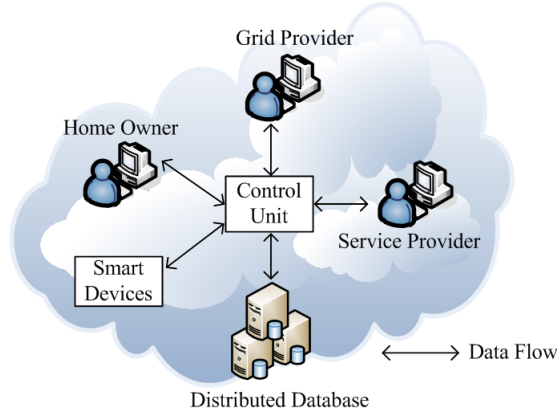


Figure 2.1: The data flows of each component in the Smart Home System.

1. What kind of data does the Home Owner keep in the Distributed Database?
2. How long will the data that is stored in the Distributed Databases be removed by itself?
3. Which data that is stored in the Distributed Databases can be reviewed by governmental officials or officers? For example, the energy consumption of the house per month or which service the Home Owner selected.
4. How can the access to the Database be controlled by a subscription mechanism? And how can this improve privacy?
5. What kind of service subscriptions does Home Owner choose from Service Providers? Each service provider may have several different subscriptions for Home Owner selection.
6. What kind of energy subscriptions does Home Owner choose from Grid Providers? Each grid provider may have several different subscriptions for Home Owner selection.

2.2 Behavioral Specification

We here use behavioral specifications (enclosed in brackets) by means of class invariants formulated over the communication history $\mathbf{h}$, which is the sequence of communication events visible to the current object. For simplicity we only consider communication events corresponding to method completion.

A completion event has the syntax $o : m(input|output)$ where o is the calling object, m the method name (which must be offered by the current interface) $input$ is the list of input parameters, and $output$ the resulting value of the call. For Unit methods we abbreviate $o : m(input|output)$ to $o : m(input)$.

Auxiliary functions are typically defined inductively over the history considering the *empty* case and $\mathbf{h}; y$ reflecting a history $\mathbf{h}$ with a specific last event y (using

semicolon for sequence append). For instance, we may define the ends with predicate by **def h endswith** $y = \text{case } h \{ \text{empty} \Rightarrow \text{false}; (h; z) \Rightarrow (z = y) \}$.

A simple, but typical example is the combination of set/get functionality for variable-like information exported through some interface.

```

interface SetGet_val {
  Unit set_val(T x)      -- set value
  T get_val()           -- read last value
  [Inv: OK(h)]          -- invariant defined over the history h,
                        -- using an auxiliary function val
  where OK(empty) = true,
        OK(h;o:set_val(x)) = OK(h),
        OK(h;get_val(|x)) = OK(h) and (x=val(h)),
        val(empty)=default<T>,
        val(h;(o:set_val(x)))=x,
        val(h;(o:get_val(|x))) = val(h) }

```

The three last equations correspond to the function definition

```

def val(h) = case h {empty => default<T>;
                  (h;o:set_val(x)) => x;
                  (h;_)=> val(h) }

```

As a useful simplification, we will introduce the convention that a pair of set/get methods with the semantics above can be abbreviated by the syntax (like a field)

```

[set-get] T val -- syntactic sugar for set-get functionality of val

```

where *val* is any name and *T* is any type, letting the annotation [*set – get*] specify that we have set- and get-functionality of *val* with the semantics above. We use the syntax of a field declaration since the abstraction simulates a visible field.

Chapter 3

Modularized Interfaces

We focus on the Control Unit with Smart Meter and its interface to the Home Owner, the Distributed Database (in a cloud), the Grid Provider (only one), Service Providers, and the Smart Devices in a smart home. (We could also have considered the Energy Provider and other Service Providers, but for simplicity we leave that out here). Fig. 3 shows the interfaces of each component of the Smart Home System.

The Grid Provider may offer a number of tariffs. The less expensive allows the Grid Provider to control essential units (by delaying the use of power of heating and charging). The more expensive tariff does not allow the Grid Provider any control inside the home. We consider three tariffs: *red*, *orange*, and *green* (allowing the most control). The Home Owner can set different style of energy usage according to how much energy he wants to save versus how much he wants to use when he needs it.

An example of a Service Provider could be a service using data from a smart refrigerator (with recognition using cameras) suggesting you what to buy based on the information about what is in the refrigerator. Another example is that provides a service using data from a smart washing machine suggesting you when to do a laundry based on time varying electricity rates of Time-of-Use (TOU) electricity rates to save money. For example, having a laundry done at night when electricity rates are low.

The different classes of smart devices (in the home) is reflected by a number of subinterfaces of the Smart Device interface. The Smart Device interface offers a number of settings depending on the actual device. For all of these the control will define three modi of usages, again called *red*, *orange*, and *green*, according to how much energy should be saved or used in flexible manner.

Modeling language

In the following we use co-interfaces, describing the interface of the *caller* object, preceded by the keyword **with**, as in the Creol language [1]

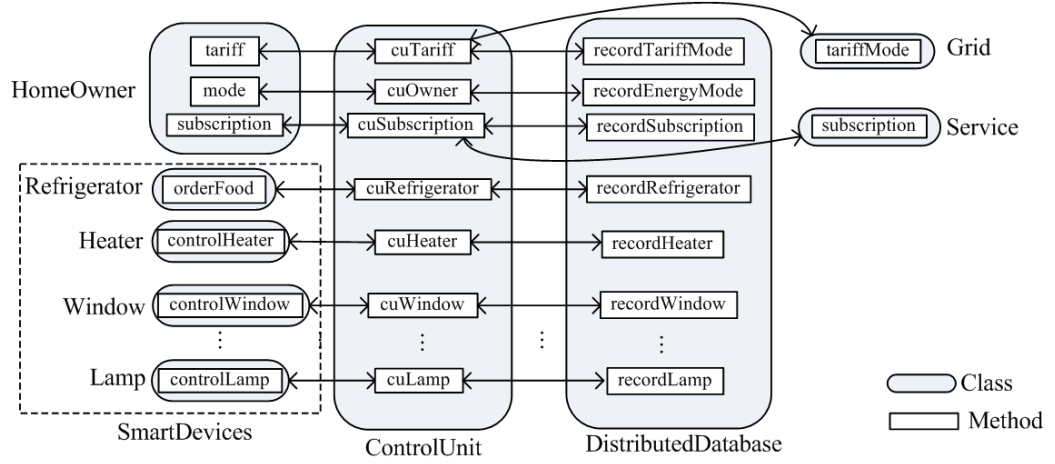


Figure 3.1: The methods of each component in the Smart Home System.

3.1 The Control Unit interface and subinterfaces

To model settings with several values, we indicate three sample values only.

```

data Load = Low | Medium | High;    -- these are sample options
data Mode = Min | Eco | Normal;      -- these are sample options
data Setting = Off | Level | On;     -- sample options
data Tariff = Green | Orange | Red;
    -- Green is cheaper but gives the Grid more control
data Flexibility = Flex | TimeFlex | Fixed;
    -- indicates how flexible a device is

data DeviceSetting = Map<Tariff><Map<Mode><Map<Load><Setting>>>;
    -- settings for a device
data HomeSetting = Map<Device><DeviceSetting>;
    -- settings for all devices
def DeviceSetting defaultSetting() = ...

```

3.2 Modeling Smart Home System

Fig. 3 shows the methods of each component in the Smart Home System, which consists of a class HomeOwner to model the Home Owner in the Smart Home, a class ControlUnit to model the Control Unit for HomeOwner to control the Smart Home, a class DBase to model the distributed database, a class Grid to model the Grid Provider in the Smart Home System, a class Service to model the Service Provider in the Smart Home System, and a set of classes SmartDevices to model the smart devices in the Smart Home System.

The ControlUnit interface has the following seven methods: cuTariff, cuOwner, cuSubscription, cuRefrigerator, cuHeater, cuWindow, and cuLamp.

```

interface ControlUnit{ with HomeOwner{
  String cuTariff(String tariff, DBase dbase);
  String cuOwner(Int id, String energyMode, DBase dbase);
  String cuSubscription(String subscribe, DBase dbase);
  String cuRefrigerator(String name, Int volume, String message
                        , DBase dbase);
  String cuHeater(Rat indoorTemp, String message, DBase dbase);
  String cuWindow(Rat outdoorWeather, String message, DBase dbase);
  String cuLamp(String switch, String message, DBase dbase);
}}

```

The method `cuTariff` is invoked to receive the tariff setting from the method `tariff` of the class `HomeOwner`, and then the `cuTariff` sends a message to the method `recordTariffMode` of the class `DBase` to make a record. The method `cuOwner` is invoked to receive the energy mode for smart home setting from the method `mode` of the `HomeOwner`, and then the `cuOwner` sends a message to the method `recordEnergyMode` of `DBase` to make a record. The method `cuSubscription` is invoked to receive the subscribe setting from the method `subscription` of the `HomeOwner`, and then the `cuSubscription` sends a message to the method `recordSubscription` to make a record. The other four methods, i.e., `cuRefrigerator`, `cuHeater`, `cuWindow`, and `cuLamp`, also do the similar function. For brevity, we focus on how the home owner controls the energy mode in the Smart Home System below.

The definition of the `ControlUnit` interface has `HomeOwner` as its *cointerface*. This ensures that only caller objects supporting the cointerface `HomeOwner` may call the methods given in `ControlUnit`. The cointerface concept comes from the Creol language, and formalizes the interface typing of the implicit caller object, and it has the advantage that it allows type-correct call-backs to the caller by methods defined in the cointerface. It no requirements to the caller objects is needed, one may use the cointerface *Any*.

```

Unit mode(ControlUnit controlunit, DBase dbase){
  Time currentTime=now();
  String energyMode="Eco"; -- Min/Eco/Normal
  Map<Int, String> energyList= EmptyMap;
  energyList= put(energyList, 1, "Min");
  energyList= put(energyList, 2, "Eco");
  energyList= put(energyList, 3, "Normal");
  energyList= put(energyList, 4, "Min");
  energyList= put(energyList, 5, "Eco");
  energyList= put(energyList, 6, "Normal");
  Rat uniMessageInterArrival=10;
  Int i=1;
  while (i<=size(keys(energyList))){
    String x = lookupDefault(energyList, i, "Error");
    currentTime=now();
    -- Time point is currentTime: Home owner chooses x energy mode.
    energyMode=await controlunit!controlUnitOwner(i, x, dbase);
    if(energyMode=="SuccessfullyInsertDBase"){
      [Cost: 2] skip;
      currentTime=now();
      -- Time point is currentTime: Home owner gets a reply that his
      -- energy mode is x.
    }else{
      -- Time point is currentTime: Home owner gets a reply that his
      -- energy mode is back to default, i.e., x.
    }
    i=i+1;
    await duration(uniMessageInterArrival, uniMessageInterArrival);
  }
}

```

In the smart home, Smart Devices can be a window, heater, washing machine, lamp, refrigerator, and so on. For each physical smart device there will be a method representing its control and communicating directly with the physical device hardware. For example, cuHeater in Control Unit can communicate with controlHeater in Heater as shown in Fig. 3.

```
interface Device {with SmartDevice {
  Unit controlDevice()
  -- the Control Unit can control Device such as turn on/off a lamp
  -- and open/close a window.
  -- Device can be a window, heater, washing machine, and so on.
}}
```

```
interface cuService {with Service {
  Unit setSubscription(Sub x)
  -- the Service Provider sets the subscription x
}}
```

```
interface cuDBase {with DBase {
  Unit insertDB(Field x)
  -- the Control Unit inserts a field x to make a record.
}}
```

3.3 The Home Owner interfaces

The Owner implements the Home Owner interface. The Owner can select the appropriate energy saving mode for his/her smart house, tariff from the Grid provider, and subscription from a Service provider. The interface of the control unit towards the real home owner (HomeOwner) is described below:

```
interface Owner {with HomeOwner {
  [set get] Mode mode; -- the Owner can set/get the energy saving mode
  -- for his/her smart home
  [set get] Tariff tariff; -- the Owner can set/get the tariff for
  -- his smart home, and Grid Provider is notified
  [set get] Subscription sub; -- owner can set/get the subscription for
  -- his smart home and the Service Provider is notified
  Unit register(Device x, Flexibility f, EnergyMap e)
  -- a new device is registered manually in the control unit
  Bool setDevice(Device x, Mode m, Load l, Setting s)
  -- adjust device control
}}
```

3.4 The Service Provider interfaces

The Service implements the Service Provider interface. The Service provides different subscriptions for the Home Owner selection. For the function of the Service we specify:

```
interface Service {with HomeOwner{
  Unit subscription()
  -- when the Home Owner sets the subscription for his/her smart home,
  -- the Service Provider is notified
}}
```

3.5 The Grid Provider interfaces

The Grid implements the Grid Provider interface. The Grid provides different energy tariffs for the Home Owner selection. For the function of the Grid we specify:

```
interface Grid {with HomeOwner{
Unit tariffMode() -- when the Home Owner sets the tariff mode for
                  -- his/her smart home, the Grid Provider is notified
}}
```

3.6 The Distributed Database interfaces

The DBase implements the Distributed Database interface. DBase will record each action of each device sent by the Control Unit. For the functions of *DBase* we specify:

```
interface DBase{ with Any {
  Bool recordEnergyMode(Field x, ...)
    -- the DBase records energy mode (field x, ...) that owner chose
  Bool recordTariffMode(Field x, ...)
    -- the DBase records tariff mode (field x, ...) that owner chose
  Bool recordSubscription(Field x, ...)
    -- the DBase records subscription (field x, ...) that owner chose
  Bool recordRefrigerator(Field x, ...)
    -- the DBase records food list (field x,...) that owner wants to buy
  Bool recordHeater(Field x, ...)
    -- the DBase records heater turn on/off that owner wants to control
  Bool recordWindow(Field x, ...)
    -- the DBase records window open/close that owner wants to control
  Bool recordLamp(Field x, ...)
    -- the DBase records lamp turn on/off that owner wants to control
}}
```

3.7 Device interfaces

The Refrigerator implements the Refrigerator interface. The Refrigerator will record what is taken in and out of the refrigerator so that it can inform the Home Owner about what kind of food he/she may need to buy. A food order is then sent to the Control Unit. For the function of Refrigerator:

```
interface Refrigerator{ with ControlUnit {
Unit orderFood(field x, ...)
  -- the Refrigerator sends food list (field x, ...), which owner
  -- wants to buy, to the Control Unit
}}
```

The Heater implements Heater interface. The Heater will be turned on/off based on the Home Owner setting. For the function of Heater:

```
interface Heater{ with ControlUnit {
Unit controlHeater(field x, ...) -- the Heater sends its action
                                -- (field x, ...) to the Control Unit
}}
```

The Window implements Window interface. The Window will be open/close based on the Home Owner setting. For the function of Window:

```

interface Window{ with ControlUnit {
Unit controlWindow(field x, ...)  -- the Window sends its action
                                   -- (field x, ...) to the Control Unit
}}

```

The Lamp implements Lamp interface. The Lamp will be turned on/off based on the Home Owner setting. For the function of Lamp:

```

interface Lamp{ with ControlUnit{
Unit controlLamp(field x, ...)  -- the Lamp sends its action
                                   -- (field x, ...) to the Control Unit
}}

```

```

data Heater = FloorHeater | WaterHeater | RoomHeater;
data KitchenDevice = Fridge | CoffeeMaker | ...
data WashingDevice = Washer | Dryer | Dishwasher;
subtype Heater WashingDevice KitchenDevice < Kind;
data Kind =    Lamp | Charger | SmartWindow | SmartSocket | ... ;

interface Device { with ControlUnitDevice{
  Bool setLevel();-- instead of Bool turnOn(), Bool turnOff()
  Nat  level();  -- current energy level
  Nat  maxEnergy; -- constant (for 100%)
  Kind kind();    -- what kind of device is this Device
                                   -- i.e. heater/lamp/charger/fridge/washer/...
}}

```

Alternatively, we could use priority.

The definition of interface Device uses ControlUnitDevice as a *cointerface*. This ensures that only caller objects supporting the cointerface ControlUnitDevice may call the methods given.

Chapter 4

Implementation

A running implementation has been worked out, and in particular the features of the smart window and the smart refrigerator has been implemented, including sending a shopping list to the home owner based on the changes in the content of the refrigerator.

For brevity, we here indicate only parts of the implementation.

```
data Setting = ....;
class CONTROLUNIT implements ControlUnit {
  -- smart meter functionality
  Nat energyUsed; -- total energy used (modulo the size of Nat)
  Nat currentEnergy; -- current amount of energy used

  Grid energy; -- the current energy Provider
  Grid grid; -- the current Grid Provider
  Load load; -- the current load level of the Grid
  Mode tariff; -- the current subscription mode (tariff)

  Owner owner; -- the current Home Owner
  Mode mode; -- the current energy saving mode as given by the owner

  Map<Device, Map<Mode, Map<Mode, Map<Load, Settings>>>> DMap;

  Unit tick(){ -- quasi code is given below:
    find energy used since last tick and recalculate overall energyUsed;
    send to database (encrypted);
    recalculate settings for each device d,
      i.e. newsetting = DMap[d][tariff][mode][load]
    updated DMap, and
    send signals to each device (if any change)
    ok :=d.setlevel(level(newsetting));
    if not ok ....}
}
```

```
class HOMEOWNER implements Owner {
}
```

```
class SERVICEPROVIDER implements Service {
}
```

```
class GRIDPROVIDER implements Grid {
}
```

```
class SMARTDEVICE implements Device {
}
```

Chapter 5

Conclusion

In order to analyze smart home systems with smart meter data stored in a cloud, we need an underlying framework allowing reasoning of systems made within the framework. Interfacing is an essential concept, which potentially can enable flexible interfaces or adaptive behavior based on dynamically added interfaces. The present approach is therefore emphasizing the use of interfaces. Another aspect is prototyping allowing experiments and early feed-back. This is supported in our approach by using an executable modeling framework. We have executed a case study with some initial experiments, with a smart home including smart windows and smart refrigerators. In this case study, the smart refrigerator sends the home owner a shopping list based on changes in the content of refrigerator.

5.1 Future Work

We may consider three different architectures to store IoT data from a Smart Home.

1. One is a decentralized architecture. In this architecture Home Owners' IoT data are stored by their storage site individually which is under control by himself/herself. Home Owner's IoT data does not have any duplicate copy in the Smart Home System.
2. Another is a multi-tier architecture. In the multi-tier architecture a leaf node represents a storage site of Home Owner, internal nodes are storage sites keeping IoT data of a group of Home Owners, which live in the same area. In this architecture, Home Owner's IoT data held by a his/her storage site will also be held by all its ancestor sites. Therefore, the root site holds all data stored in the Smart Home System.
3. The other is a centralized architecture that is stored by the central database of the Smart Home System. There is no storage site in Home Owner's house.

We would then like to compare and discuss these three architectures of the Smart Home System with respect to the following three metrics: *performance*, *privacy*, and *security*. The simulation results can possibly enable managers and Home owners to comprehend the influences of different storage deployments

on performance, privacy, and security issues in the Smart Home System. The results also can help them to choose a proper storage deployment and a Smart Home System structure to achieve better system performance, higher privacy, and stronger security.

In the experiment, we will create a normal access IoT data flow with *random attacks* for the different architectures to discuss the security and privacy issues of the system. These attacks might cause the system failure, missing data, or leaking sensitive data. After that we will know which architecture has better system performance, higher privacy, and stronger security in the these attacks. The results might show that each architecture has its pros and cons. No one is perfect.

Concerning attacks, we can assume that IoT data having 0.1% probability will be decrypted when it is transmitted to database such that IoT data need to be retransmitted when IoT data missing, IoT data lost when IoT data storage devices are broken, or IoT privacy data is exposed by malicious attacks. Based on these situations, we can evaluate security metric and privacy metric in each architecture of the Smart Home System.

Acknowledgment

This work was supported by the project *IoTSec* of the Norwegian Research Council.

Bibliography

- [1] Einar Broch Johnsen, Olaf Owe: An Asynchronous Communication Model for Distributed Concurrent Objects, *Journal of Software and Systems Modeling* 6(1): 39-58, Springer 2007.
- [2] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, Martin Steffen, ABS: a core language for abstract behavioral specification, *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, November 29-December 01, 2010, Graz, Austria
- [3] K. T. Weaver. A perspective on How Smart Meters Invade Individual Privacy. *Technical report, SkyVision Solutions, Naperville, Illinois*, 2014.
- [4] Einar Broch Johnsen, Olaf Owe, Rudi Schlatte, Silvia Lizeth Tapia Tarifa: Validating Timed Models of Deployment Components with Parametric Concurrency. *Postproceedings of International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010) Lecture Notes in Computer Science* 6528, pp. 46-60, 2011.
- [5] Bill Lichtensteiger et al. RF Mesh Systems for Smart Metering: System Architecture and Performance. *First IEEE International Conference on Smart Grid Communications (SmartGridComm 2010)*. IEEE, 2010.