What to Do When Requirements Are Changing All the Time?
A Control System Example

Bert de Brock(✉)

Faculty of Economics and Business, University of Groningen, PO Box 800, 9700 Groningen, AV, The Netherlands
E.O.de.Brock@rug.nl

Abstract. In our earlier work we sketched an approach to developing software systems. The goal of this paper is to further illustrate the applicability and use of that approach. Via the practical example of the development of a control system, we illustrate the applicability of our approach to another type of system (other than the usual information system) and the **manoeuvrability** ('agility') of our *textual* System Sequence Descriptions. We discuss how to deal with situations where requirements are changing all the time ('agility' during requirements analyses). We also want to sketch the **mental process** of going from a simple, naïve software solution towards various more subtle ones, probably inspired/ guided by brainstorms with customers. In this case, it even ends up in a **generic system** (so, not for one particular user organization only).

Keywords: Changing requirements · Agility · Textual System Sequence Description · Controller · Controlled system · Generic system · System scope

1 Introduction

When the requirements change all the time, the question is: How to deal with all those changes concretely in practical situations? We will zoom in on that problem. In order to really understand the problems when requirements are changing all the time and how to master them, we have to show the nitty-gritty details as well, because managing them in such constantly changing circumstances makes it all so difficult. The manoeuvrability of our *textual* System Sequence Descriptions (tSSDs) turns out to be very helpful when the requirements are changing all the time ('agility' of our textual SSDs).

This paper is also meant to illustrate and work out the mental process of going from a simple, naïve solution towards various more subtle ones, probably inspired/guided by brainstorms with the customer ('agility' during requirements analysis).

In our earlier work we sketched an approach to developing software systems [1, 2]. Comparisons to other work are already made in [2], in this volume. But the used examples might suggest that the approach applies to 'information systems' only: There was little to no interaction with other systems in those examples. So the question arose whether our approach is applicable to other types of systems as well, e.g., control systems. Yes, it is, as we will explain and show in this 'companion' paper.

B. Shishkov (Ed.): BMSD 2021, LNBIP 422, pp. 317–329, 2021.
https://doi.org/10.1007/978-3-030-79976-2_20

A **control system** (*controller* for short) is a system that has to manage the behaviour of other systems. So, a controller typically has (much) interaction with other systems. Our running example concerns thermostats, being classic examples of control systems. The running example also illustrates the phenomenon that the data structures of a system are usually more stable than the processes that system has to support.

With this paper we also want to discuss and illustrate the 'scope issue': What should be inside and what outside the scope of the system? Finally, we want to illustrate the development of a *generic* system, not a system for one particular user organization only.

The rest of the paper is organized as follows. Section 2 presents the preliminaries needed for understanding the rest. Section 3 gives an initial, simple, concrete description of our running development example. Section 4 introduces more than ten extensions, variants, and/or alternative options (except Sect. 4.3, which zooms in on the data needed). Section 5 gives an overview and the paper ends with a contribution section. For the reader's convenience, the Appendix shows the finally resulting textual SSDs.

2 Preliminaries

In [1] a grammar for textual SSDs is proposed. It is similar to the grammar for a programming language, except for the atomic instructions. We recall a part of that grammar below. The terminals are written in bold. The nonterminal $\underline{\text{A}}$ stands for 'atomic instruction' (or *step*), $\underline{\text{P}}$ for 'actor' (or *participant*), $\underline{\text{M}}$ for 'message', $\underline{\text{S}}$ for 'instruction' (or *SSD*), $\underline{\text{C}}$ for 'condition', $\underline{\text{N}}$ for 'instruction name', and $\underline{\text{D}}$ for 'definition':

A ::= P → P**:** M /* *where* 'X → Y: M' *means:* 'X sends M to Y'
S ::= A | S **;** S | S **,** S | **begin** S **end** | **if** C **then** S **endif** | **for each** <set element> **do** S **end**
  | **do** N
D ::= **define** N **as** S **end**

where '$\underline{\text{s1; s2}}$' means 'first do s1, then do s2', '$\underline{\text{s1, s2}}$' means 'do s1 and s2, in any order'. The construct '**do** N' is known as an *Include* or a *Call*. Definitions can be parameterized (see Sects. 4.7 and 4.8 for examples). The values for nonterminals P, C, M, and N are application dependent ('domain specific') and will appear naturally during the development of the specific application. We will sometimes use the terminal **System** for P to represent the system under consideration.

In order to avoid ambiguity, we use the binding rule that '**,**' binds stronger than '**;**'. We can break through this standard reading by using the 'brackets' **begin** … **end**.

For atomic instructions we can distinguish the following four situations:

(a)	Actor → **System:** i	Indicates the input messages the system can expect
(b)	**System** → **System:** y	Indicates the transitions (or checks) the system should make
(c)	**System** → Actor**:** o	Indicates the output messages the system should produce
(d)	Actor → Actor2**:** x	A step outside the system (maybe useful for understanding)

where Actor ≠ **System** and Actor2 ≠ **System** (but Actor and Actor2 might be the same). If the participant before and after the ' →' are the same, the atomic instruction indicates what that participant has to do himself/herself/itself. We call step (a) an *input* step, (b) an *internal* step, (c) an *output* step, and (d) an *external* step.

The question arose whether our approach with textual SSDs only works in the context of 'information systems' or also in the context of control systems, for instance. In an 'information system' it is not uncommon that during a session (or Use Case) there is one fixed actor (role) interacting with the system. In other words, a 'dialogue' (bilateral conversation) between the actor and the system only.

A **control system** (*controller* for short) is a system that has to manage the behaviour of other systems, often triggered by signals coming from outside. A controller typically has (many) interactions with other systems, both on the input side as well as on the output side. In the context of controllers, typical atomic interactions are:

Controller → Sensor**:** Request
Sensor → Controller**:** Signal
Controller → Controlled System**:** Command
Controlled System → Controller**:** Feedback

A sensor might send a signal to the controller all by itself, without a previous request. The interaction steps shown in a picture (with several sensors and controlled systems) (Fig. 1):

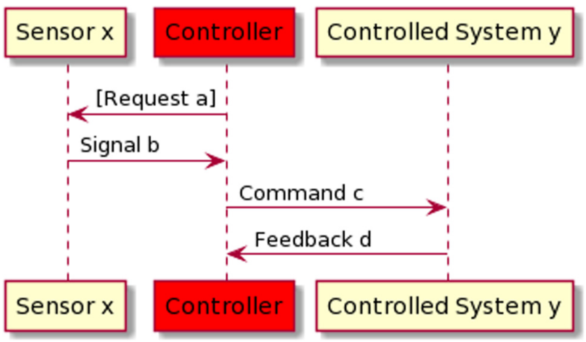


Fig. 1. Typical atomic interactions with a controller

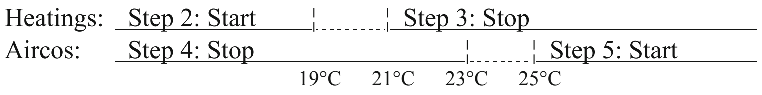
3 Initial Description of the Running Development Case

Suppose we have a building with several rooms (e.g., an office or a school). Rooms have sensors for measuring the temperature (‘temp.’ for short) and might also have heatings and air conditioners (aircos). The systems to be controlled in this example are the heatings and aircos. The system to be developed (simply called ‘the system’) must be able to receive temp. measurements from the sensors and, when needed, start or stop the heatings or aircos in that room. (So the system is a type of ‘distributed thermostat’.)

To be more precise, the heating(s) in a room must be started when the temp. in that room drops below 19 °C and must be stopped when that room temp. comes above 21 °C and, similarly, the airco(s) in a room must be started when that room temp. comes above 25 °C and must be stopped when that room temp. drops below 23 °C. In that case, a simple (even naïve) version of the main Use Case, *Handle Measurement*, might be:

1. A sensor sends a measured temperature to the system.
2. If that temp. is below 19 °C then the system starts the heating(s) in that room.
3. If that temp. is above 21 °C then the system stops the heating(s) in that room.
4. If that temp. is below 23 °C then the system stops the airco(s) in that room.
5. If that temp. is above 25 °C then the system starts the airco(s) in that room.

So, for any measured temp. one or even two of the steps 2–5 apply, as depicted below:



We note that in this example the sensors are not considered part of the system to be developed. In other words, they are outside the *scope* of the system.

In Sect. 4 we introduce many extensions, variants, and/or alternative options. Usually we first try to formulate them for controllers in general and then make it more specific for our distributed thermostat.

4 Subsequent Extensions, Variants, and Alternative Options

4.1 No Unnecessary Commands

If upon receipt of a measurement, a controlled system (CS for short) is already in the desirable state then the controller does not need to send a command to that CS anymore. Expressed in the form of a Use Case:

1. A sensor sends a signal/measurement to the controller.
2. If that signal might call for action *and the relevant CS is in undesirable state* then the system sends the proper command to that CS.

And schematically in the form of a textual SSD:

```
1. Sensor → Controller: Signal;
2. if signal might call for action and CS is in undesirable state
   then Controller → Controlled System: Command endif
```

We note that the combination of Step 1 and Step 2 constitutes a so-called ECA-rule for our controller (Event, Condition, Action): *Triggering Event; if Condition then Action.* Expressed as a textual SSD:

```
1. X → System: Signal ;                              /* Triggering Event
2. if Condition then System → Y: Command endif       /* if Condition then Action
```

The idea of no unnecessary commands leads to the next version of *Handle Measurement* for our distributed thermostat from Sect. 3. In the form of a Use Case:

1. A sensor sends a measured temperature to the controller.
2. If that temperature is below 19 °C *and heating(s) in that room are 'Off'* then the controller starts those heating(s).
3. Similarly, if above 21 °C *and heating(s) are 'On'* then stop those heating(s).
4. Similarly, if below 23 °C *and airco(s) are 'On'* then stop those airco(s).
5. Similarly, if above 25 °C *and airco(s) are 'Off'* then start those airco(s).

And schematically in the form of a textual SSD (using variables instead of pronouns):

```
1. Sensor x → Controller: Temperature t ;
2. if t < 19 °C then for each heating h in the room of x that is 'Off'
                     do Controller → h: 'On!' end  endif ,
3. if t > 21 °C then for each heating h in the room of x that is 'On'
                     do Controller → h: 'Off!' end  endif ,
4. if t < 23 °C then for each airco a in the room of x that is 'On'
                     do Controller → a: 'Off!' end  endif ,
5. if t > 25 °C then for each airco a in the room of x that is 'Off'
                     do Controller → a: 'On!' end  endif
```

The commas between steps 2, 3, 4, and 5 say that those steps can be done in any order.

4.2 Constants Should Become 'Adjustable'

System requirements might contain constants which, as might turn out only later, should be adjustable. That means that those constants should be replaced by (system) variables.

4.2.1 Variable Thresholds

As presented now, the concrete threshold temperatures (19, 21, 23, and 25 °C) might end up 'hard-coded' in the thermostat, our control system under development. However, as a new user requirement, these thresholds should become adjustable. Therefore we introduce the four variables Hmin, Hmax, Amin, and Amax representing the minimum and maximum thresholds for the heatings and aircos, respectively. We also add the condition that Hmin $\leq$ Hmax $<$ Amin $\leq$ Amax.

In Step 2 of the most recent textual SSD, t $<$ 19 °C must be replaced by t $<$ Hmin; similarly, t $>$ 21 °C by t $>$ Hmax in Step 3, t $<$ 23 °C by t $<$ Amin in Step 4, and finally t $>$ 25 °C by t $>$ Amax in Step 5. The temperatures 19, 21, 23, and 25 °C could serve as *default values* upon installation.

4.2.2 Variable Thresholds Per Room

On hindsight, not all rooms do need the same threshold temperatures. E.g., a corridor might have a minimum threshold of 17 °C instead of 19 °C. Now we need these four threshold values *per room*, each with the condition that Hmin $\leq$ Hmax $<$ Amin $\leq$ Amax. The advantage is that the thresholds can now be set per room.

4.2.3 Variable Thresholds Per Room Type

That the thresholds must be set per room turned out to be a disadvantage in case of large buildings. Another variant is that the thresholds only depend on the *type of room* (e.g., classroom, gym hall, corridor, etc.), not on the individual room. In that case we need those four threshold values *per room type*, now of course with the condition *per room type* that Hmin $\leq$ Hmax $<$ Amin $\leq$ Amax. The advantage is that the thresholds can now be set uniformly for all rooms of the same type.

4.3 Which Data Does the Controller Need?

It is time to see which data (structure) the controller needs. In general, the controller needs to 'know' the *configuration*: the sensors, the controlled systems, and their state.

4.3.1 Configuration Data and State Data

In our running example: The thermostat needs to 'know' the sensors, the heatings, the aircos, their states, the rooms they are in, and the type of rooms (see Sect. 4.2).

Concretely: Suppose that each $\underline{\text{sensor}}$ has a unique sensor ID (SID), each $\underline{\text{heating}}$ has a unique heating ID (HID), each $\underline{\text{airco}}$ has a unique airco ID (AID), each $\underline{\text{room}}$ has a unique room ID (RID), and each $\underline{\text{room type}}$ has a unique room type ID (RTID). Furthermore, the controller needs to know the *state* of each heating and of each airco. Moreover, the controller needs to know the *room* of each sensor, heating, and airco. The controller also needs to know the *room type* of each room. And, in case of Sect. 4.2.3, the controller needs to know those four thresholds for each room type and also needs to know the condition Hmin $\leq$ Hmax $<$ Amin $\leq$ Amax per room type.

For Sect. 4.2.3 this leads to the following concepts and attributes (where we indicate the identifiers by a '!' in front and the referencing attributes by a '^' in front):

Sensor:	! SID, ^ RID
Heating:	! HID, ^ RID, State ('On' or 'Off')
Airco:	! AID, ^ RID, State ('On' or 'Off')
Room:	! RID, ^ RTID
Room Type:	! RTID, Hmin, Hmax, Amin, Amax

with the condition per room type that
Hmin $\le$ Hmax $<$ Amin $\le$ Amax (Fig. 2).

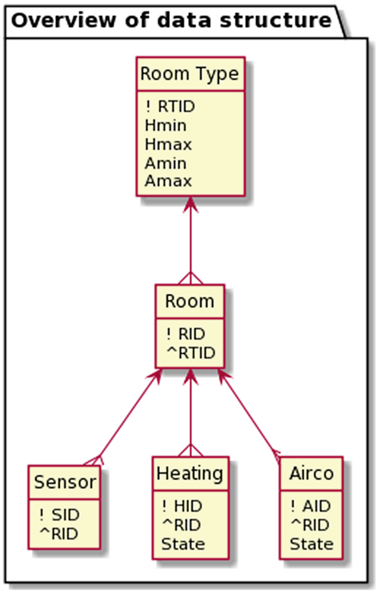


Fig. 2. Overview of needed data

4.3.2 Keeping the Data Up-to-Date

The controller must keep its data up-to-date. So, when it changes the state of a controlled system, the controller has to update that state in its own registration as well. E.g., with variable thresholds per room type, Step 5 in the tSSD in Sect. 4.1 now becomes (with the update step underlined):

```
if t > Amax of the type of room sensor x is in
then for each airco a in the room of x that is 'Off'
    do Controller → a: 'On!' ;
       Controller → Controller: Register a as 'On'
    end
endif
```

4.3.3 Remembering the Measurements Too?

Controllers handle an incoming measurement by taking the appropriate actions. After that, the system can 'forget' that measurement. But, after all, the user organization (and the producers/installers of the controlled systems) wanted to be able to look at the past measurements. In that case, the incoming measurements must be remembered too, together with a timestamp. Then we need one more concept, say 'Measurement'.

For our running example this implies the following attributes for 'Measurement' (where the underlined combination *SID, Timestamp* is uniquely identifying):

Measurement: SID, Timestamp, RID, Temperature.

We did not indicate SID or RID as *referencing* attributes because they might refer to 'old' sensors or rooms that do not exist anymore.

Our main use case *Handle Measurement* now gets an extra step: Step 1 is replaced by

1. A sensor sends a measured temperature plus timestamp to the controller.
2. The controller stores the info of the sensor, its room, measured temp., and timestamp.

And schematically, as a textual SSD (where r_x indicates the room sensor x is in):

1. Sensor x $\rightarrow$ Controller: Temperature t plus timestamp s**;**
2. Controller $\rightarrow$ Controller: Store sensor x, room r_x, temperature t, and timestamp s

4.4 External Data Store

The measurements could be stored inside the system under development or in a separate external system (making the architecture less monolithic).

There could be several reasons for that:

– the organization/installer/producer already developed a system for that or has a system filled with measurement data with which 'our' data must be combined
– the measurement data must be integrated with other external data (e.g., weather data)
– the controller gets overloaded in case of heavy trend analyses on its measurement data

Whatever the reasons are, in this way we move more towards a micro-service architecture (https://www.guru99.com/microservices-tutorial.html). The new Step 2 would become:

2. Controller $\rightarrow$ External system: Store sensor x, room r_x, temp. t, and timestamp s

4.5 Simple Sensors Cannot Provide a Timestamp

In the example until now, it was the sensor that provided the timestamp (the time of *measurement*). If the sensors are so simple that they cannot provide a timestamp, the system itself could add a timestamp (say, the time of *receipt*) by using its internal clock. In that case, the first two steps in the use case become:

1. A sensor sends a measured temperature to the controller.
2. The controller stores the sensor info, its room, measured temp., and time of *receipt*.

And schematically, as a textual SSD (where s' is the time of *receipt*):

1. Sensor x $\rightarrow$ Controller: Temperature t**;**
2. Controller $\rightarrow$ Controller: Store sensor x, room r_x, temperature t, and timestamp s'

So, now the timestamp pops up in the second step, provided by the controller itself.

In case of an external data store, the second occurrence of 'Controller' in the second step can be replaced by 'External system' (cf. Sect. 4.4).

4.6 Synchronous Feedback from a Controlled System

It turned out to be a flaw that the controller adapts the registered state of a controlled system as soon as it issued such a command to that controlled system, without knowing whether the intended state actually changed. An extension/improvement/option is that the controlled system gives feedback about its status to the controller. Only then, the controller would change the registered state of that controlled system. In that case, the last four steps in the tSSD in Sect. 4.1 must be adapted. E.g., with variable thresholds per room type, Step 5 in the tSSD now becomes (with the new parts underlined):

```
if t > Amax of the type of room sensor x is in
then for each airco a in the room of x that is ‘Off’
      do Controller → a: ‘On!’ ;
         a → Controller: NewState(a) ;
         Controller → Controller: Register NewState(a) as the new state of a end
endif
```

4.7 Asynchronous Feedback from a Controlled System

Often it takes a (tiny) while before a controlled system gives feedback. Meanwhile, the controller has to do other things as well… So, another option is that the controller does not wait for an answer and only adapts the registered state once the status feedback from the controlled system comes in. That leads to another UC, say *HandleStatusFeedback*:

1. A controlled system sends its status to the controller.
2. The controller adapts the registered status of that controlled system accordingly.

And the corresponding tSSD, cast in the form of a parameterized definition:

```
define HandleStatusFeedback(y) as
  y → Controller: State(y) ;
  Controller → Controller: Register State(y) as the state of y
end
```

The last four steps in the main UC (*Handle Measurement*) should be without these two instructions now. Step 3, for instance, now becomes:

```
if t < Hmin of the type of room where sensor x is in
then for each heating h in the room of x in state ‘Off’
      do Controller → h: ‘On!’ end
endif
```

We note that it is easy to go back from the asynchronous to the synchronous situation, since we can just call *HandleStatusFeedback* within this most recent version of Step 3:

```
if t < Hmin of the type of room where sensor x is in
then for each heating h in the room of x in state ‘Off’
      do Controller → h: ‘On!’ ;
         do HandleStatusFeedback(h)
      end
endif
```

4.8 Scheduled or Even Dynamic Threshold Changes

The user organization subsequently indicated that it is useful to have lower threshold temperatures for the heatings at night than during daytime. For example, in an office say 21 °C from nine till five from Monday till Friday, and 15 °C during the other time periods. And such schedules can be much more subtle, of course. It could be a wish that the controller changes the thresholds automatically in such cases. This could be realized if the controller 'knows' the schedule and has an internal clock. Then the question came up: Which type of schedules for threshold changes should be possible? E.g., based on the combination of weekday and time during the day only (as in the example above)? And thereafter the (internal) discussion in the organization advanced even more: Ideally, threshold changes might be determined dynamically, e.g., based on external conditions or events. For instance, in order to have a temp. of 21 °C at 9:00, it might be necessary to start the heatings (much) earlier, but how much earlier can depend on the inside temperature at hand, the outside temperature, the size (and isolation) of the room to be warmed, etc. Gradually the question arose whether the controller itself should know the schedule or that an (intelligent) external system should trigger the system with new threshold temperatures at the right moments. This idea was partly inspired by the rise of systems such as Homey (https://homey.app/en-gb/) where its users can constitute all type of rules to turn down (or off) the thermostat. This option is more flexible and, moreover, that external system could easily be replaced by a more subtle/advanced/ sophisticated one (provided that it keeps the same type of interface-mechanism with our controller). Where the development started with a concrete controller, by now they are thinking of a **generic** COTS-system (*Commercial off-the-shelf*) to be sold on the market, not meant for one particular customer anymore…

We will work out this generic option. So in other words, the scheduling/scheduler will be considered outside the *scope* of the system under development.

In conclusion, an external system - but also a human being - should be able to trigger our system with new threshold temperatures. We will use the term 'thresholder' here (instead of 'user'). For a change, we suppose that the threshold adaptions are on the level of individual rooms, not on the level of room types. So, Sect. 4.2.2 applies, not Sect. 4.2.3. Consequently, the attributes Hmin, Hmax, Amin, and Amax move from Room Type to Room in the model in Sect. 4.3.1, now with the condition *per room* that Hmin $\leq$ Hmax $<$ Amin $\leq$ Amax. The use case *AdaptThreshold* could run as follows:

1. A thresholder sends a (new) value for a threshold of a certain room to the controller.
2. The controller adapts the value for that threshold for that room.
3. The controller asks the sensor in that room for the current temperature.

The UC does not need to continue any further here, because once the controller gets the temp. from the sensor, the 'old' UC *Handle Measurement* starts and the controlled systems will be commanded accordingly if necessary, based on the new threshold value.

Note that there are four types of thresholds per room: Hmin, Hmax, Amin, and Amax. With A $\in$ {Hmin, Hmax, Amin, Amax}, we define a parameterized *AdaptThreshold*:

```
define AdaptThreshold(A, r, t) as
  Thresholder → Controller: Update threshold A in room r to temperature t ;
  Controller → Controller: Update threshold A in room r to temperature t ;
  for each sensor x in room r do Controller → x: SendTemperature end
end
```

As explained before, hereafter the controller will receive the temperature from the sensor(s), then *Handle Measurement* starts and the controlled systems will be started or stopped accordingly (if necessary).

If threshold adaptions were on the level of room types, as in Sect. 4.2.3, then we should have to change '*in room r*' by '*for room type r*' in the first two steps of the tSSD and by '*in a room of type r*' in the last step of the tSSD.

5 Interactions Overview: Our Controller and Its Environment

We end with a general overview of the typical interactions between our controller and its environment. Instead of the earlier terms 'External System' or 'External Data Store' we use 'Data Store' because it might or might not be part of the system to be developed. The variables x and y in the overview indicate that there can be several such actors (Fig. 3):

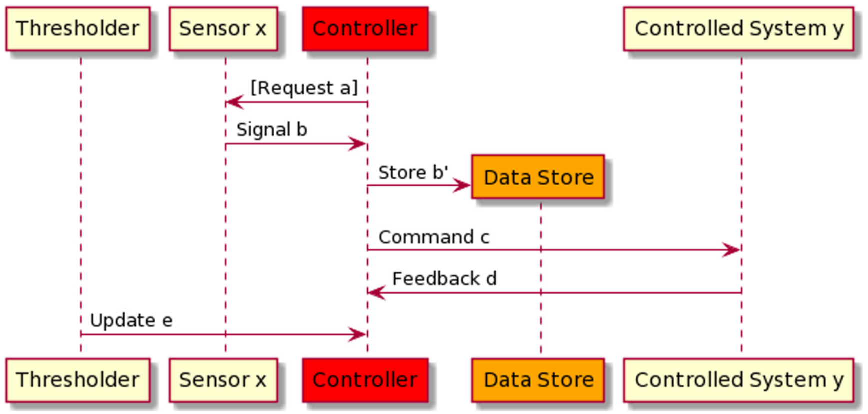


Fig. 3. The typical interactions between our controller and its environment (This figure is not meant as a sequence diagram)

6 Contribution

As promised in the abstract, the paper discussed and illustrated the following topics:

– When system requirements are **changing all the time**: How to deal with that? As illustrated throughout the paper, by usually writing down only the (small) differences with a previous solution, and not writing out the new situation completely. Writing

out the situation completely might be done once you have a good overview - that is why Sect. 4.3 came so late – or, contrarily, when you lost your overview.
- The (**mental**) **process** of going from a simple, naïve software solution towards various more subtle ones, probably inspired/guided by brainstorms with customers, was richly illustrated by the (>10) extensions, variants, and/or alternative options we introduced
- What should be inside and what outside the system scope? The **scope issue** was illustrated by the Data Store issue and the scheduling/scheduler discussion
- The development started with a concrete controller but ended with a **generic** COTS-system, not meant for one particular customer anymore (Sect. 4.8)
- **Agility** during requirements analyses was shown over and over again with all those extensions, variants, and/or alternative options we discussed
- The feasibility of our approach to another type of system, a **control system**, was illustrated throughout the paper
- The **manoeuvrability** ('agility') of our textual SSDs was shown during the many discussions of all types of variants ('textual SSDs in operation')

Acknowledgment. The author wants to thank Wilco Wijbrandi from TNO Research for the fruitful discussions we had about the many possible variants concerning thermostats.

Appendix: The Resulting Textual SSDs

The tSSD HandleMeasurement below is the version with variable thresholds per room type (Sect. 4.2.3), sensors that provide a timestamp (Sect. 4.3.3), a data store for registering measurements (Sect. 4.4), synchronous feedback from the heatings (Sect. 4.6), and asynchronous feedback from the air conditioners (Sect. 4.7). The tSSD HandleStatusFeedback originates from Sect. 4.7 and the tSSD AdaptThreshold from Sect. 4.8. (Blue underlined words serve as links too.)

```
define HandleMeasurement(x, t, s) as
o  Sensor x → Controller: Temperature t plus timestamp s ;
o  Controller → Data Store: Store sensor x, room r_x, temperature t, and timestamp s ,
o  if t < Hmin of the type of room sensor x is in
   then for each heating h in the room of x that is 'Off'
          do Controller → h: 'On!' ;
             do HandleStatusFeedback(h)
          end
   endif ,
o  if t > Hmax of the type of room sensor x is in
   then for each heating h in the room of x that is 'On'
          do Controller → h: 'Off!' ;
             do HandleStatusFeedback(h)
          end
   endif ,
o  if t < Amin of the type of room sensor x is in
   then for each airco a in the room of x that is 'On'
          do Controller → a: 'Off!' end
   endif ,
o  if t > Amax of the type of room sensor x is in
   then for each airco a in the room of x that is 'Off'
          do Controller → a: 'On!' end
   endif
end

define HandleStatusFeedback(y) as
   y → Controller: State(y) ;
   Controller → Controller: Register State(y) as the state of y
end

define AdaptThreshold(A, r, t) as
   Thresholder → Controller: Update threshold A in room r to temperature t ;
   Controller → Controller: Update threshold A in room r to temperature t ;
   for each sensor x in room r do Controller → x: SendTemperature end
end
```

References

1. De Brock, E.O.: On system sequence descriptions. In: M. Sabetzadeh, et al. (eds.) Joint Proceedings of REFSQ-2020 Workshops, etc. Pisa, Italy (2020)
2. De Brock, E.O.: From Elementary user wishes and domain models to SQL-specifications. In: Shishkov, B. (eds.) Business Modeling and Software Design. BMSD 2021. Lecture Notes in Business Inf. Processing, vol. 422, pp. 97–117. Springer, Cham (2021)