Mats Neovius

Formal Stepwise Development of an In-House Temperature Control System

Turku Centre for Computer Science

TUCS Technical Report
No 1078, May 2013

Formal Stepwise Development of an In-House Temperature Control System

Mats Neovius

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
mats.neovius@abo.fi

TUCS Technical Report
No 1078, May 2013

Abstract

With the possibility to sense and control the physical environment, restrictions and demands on how it is controlled may be defined. The system guaranteeing the stated conditions by analysing the sensory data and triggering a corrective action is called a control system. Formal verification ascertains the correct mathematical functionality of such a control system given a model of the environment. Moreover, formal specification facilitates the engineers' understanding of the system features. In this report, we formally specify a temperature control system responsible for maintaining a certain in-house temperature in the Event-B language. We verify this specification in the Rodin Platform tool. The specification is developed independently in a stepwise manner to be suitable for integration with other control systems.

Keywords: Formal modelling, Event-B, refinement, temperature control, control system, adaptive house

TUCS Laboratory
Distributed Systems Laboratory

1. Introduction

Actuators of a control system are implemented to serve for a task. In this paper reporting on the development of an in-house temperature control system (*tcs*), we consider the actuators to be radiators and air-conditioning units that ubiquitously adjusts the temperature of a designated space indoors. The motivation is that in residential buildings, the in-door air temperature control system consumes a significant amount of the total energy consumed by residential buildings. Hence, managing this control system provides means for the topical strive for increased energy-awareness and energy-preservation; controlling the actuators adjusting this physical phenomenon is of central interest.

The functionality of the *tcs* is to switch heating and cooling units on and off for adjusting the temperature. The aim is to be able to prove that, using the actuators, the temperature will reside within given bounds, namely the 'low' and 'high' thresholds. Fundamental conditions to prove include those of not having a heating unit and a cooling unit switched on simultaneously. In addition, we develop the specification to also model the case when the residents are on "vacation" allowing the thresholds to be relaxed by disregarding comfortability. Altogether, the development consists of an initial view to which we gradually add details in four (4) revision steps. To manage the complexity and for defining the system boundaries unambiguously, this paper reports on a formal development of a system alike that described above.

For the means to formalise there is an abundance of languages, often referred to as *formal methods*. Roughly speaking, the formal methods can be divided into three groups: one focusing on communicational matters known as event-based formalisms [19] [22] [23] [7]; one focusing on the state of a software known as state-based formalisms including [9], [3] [6] [21] [18] [15] [2]; and one focusing on properties (typically temporal logics) [26] [16]. Which of all of the formal methods is selected is typically dictated by need, the author's familiarity with its semantics, possible tool support and other related features.

In this paper, we use the Event-B [2] state-based formal method as the formal method in which to specify *tcs*. This selection is motivated by the convenience of modelling the states of heating and cooling units of the *tcs*, the author's familiarity with the language's semantics, its tool support as well as by its support of refinement. The tool support for Event-B [2] used in this paper is Rodin Platform tool [17] [5] featuring automatic theorem provers. Event-B also supports the *refinement methodology* [8] [24], a means to introduce details in a stepwise manner without compromising correctness. This is based on the refinement calculus framework that in turn bases on lattice theory [12] to rigorously define a relation between the "abstract" and "concrete" specification. The Rodin Platform tool [17] [5] theorem provers also supports checking conditions of refinement relations, i.e. the tool proves the refinement relations correct.

This paper is outlined as follows: In Section 2 we briefly introduce the Event-B language followed by a short introduction to the Rodin Platform tool and the conditions we aim to show. Section 3 describes the initial model followed by a refined model in each of its subsections. These refinement steps are outlined in Figure 2. Section 4 discusses some limitations with respect to the model and Section 5 concludes the paper.

2. Formal Preliminaries

For something to be formal, this means that it follows or is defined in accordance to some rule(s). When expressing something in a programming language, that is a formal representation for defining how some task is performed; when formalising something as in this paper, it has to do with defining the boundaries within which the execution of the system (that is to be specified) resides. Hence, a formalisation defines *what*, not *how*; and what from a perspective of boundaries. Within these boundaries and with the formal notation of the language in which the specification is expressed, some features may be proved to hold.

The closeness of the specification with the envisaged real implementation is central. However, for mathematical-logical intentions on ability to prove correctness, we will *model* this "real" environment including all its uncertainty and inconsequence in a precise manner, i.e. we will disregard inconsequent behaviour such as actions of a human operator. Hence, we generalise the application-specific environment by assuming it to behave rationally, so any formal specification may regard only the rational mathematical-logical (software) part.

With mathematical correctness of the specification, we may claim our system specification faultless. Thus, if such a system fails, this is due to having modelled the environment too far from the real implementation environment. This is equivalent to having approximated the environment falsely or having disregarded some part of it. However, analysing uncertainties of the model falls outside the consideration of this paper. Hence, we excuse ourselves to make this necessary assumption for the sake of correctness.

2.1. The Event-B Language and the Rodin Platform

Event-B is a language for state-based formal specification. The state space of the specification is bound by its variables' interval and its constants. The momentarily state is defined by the values of these variables and constants at a given moment. These system variables' values may be updated by transitions, the so-called *event*s. Each event is a guarded atomic set of concurrent variable updates, i.e. their execution is atomic. The event *guard* is a predicate on the variables that when true, enables the transitional part of an event for execution. The transitional part of an event is typically called the event's *action*. These actions support non-deterministic assignment of variables. When several events' guards evaluate true (events are *enabled*), one is chosen non-deterministically (demonically) for execution. On a specification level, the execution stops when no event is enabled, in case the execution may not proceed. Whether or not this condition is desired is task-specific, i.e. it can model a deadlock or a desired termination condition.

Properties of the specification may be declared as invariants. An invariant is a condition that holds initially and after the execution of any event. Correctness of a specification is then subject to showing the integrity of the invariants. The semantics of the language is described using *before-after* (*BA*) predicates [2] [4].

Fundamentally, a specification in the Event-B language consists of two components: a *context* and a *machine*, see Figure 1. The context defines the static part of the specification by declaring *constants* and *carrier sets* whose properties are *axiom*ised for the specification. A context may be *seen* by a machine, in which case the machine may use the constants. The machine defines the variables, invariants and the events of the specification. Hence, the machine declares the actions whose preservation of the invariants declares the specification consistency, i.e. its correctness. Effectively this means that each invariant (predicate) needs to evaluate true in every reachable state.

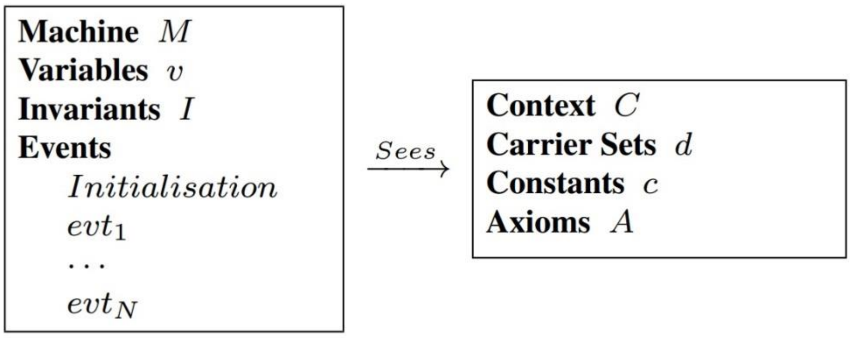

Figure 1: A machine *M* and a context *C*

The Event-B formal method has tool support in form of the Rodin Platform tool [17] [5]. The Rodin Platform tool employs a proof manager that automatically generates proof obligations. Proof obligations are statements that need to be proved true for the specification to be correct. These proof obligations are logical sequents that have to do with the invariants and the refinement relation between specifications. Each proof obligation may either be automatically discharged by the integrated theorem provers, may require interaction from the developer or, simply, be non-provable, i.e. false.

2.2. System Development in Event-B

A specification in any language outlines the domain of discourse. In case of Event-B this consists of two components *context* and *machine* that when composed in parallel, denoted ||, constitute the specification. Criterions for parallel composition include non-overlapping variable names. If there are overlapping variable names, this may trivially be solved by simple renaming. Reversely to parallel composition, a feature of a specification in Event-B is the possibility for decomposition. Hence decomposition of M to M_1 and M_2 is denoted $M = M_1 \parallel M_2$. Decomposition is implemented by either sharing variable(s) or event(s), originally brought forth by Sere [28] in the Action Systems framework; and later implemented in Event-B by Abrial [1] and Butler [13]. The Rodin Platform tool features a decomposition plugin [14].

In addition to parallel composition and decomposition, the components of the Event-B language are subject to refinement. Refinement, denoted $\sqsubseteq$, is a relation that provides a means for stepwise introduction of details to a component. The goal with development alike is to reach a specification at a level of concretisation sufficient to define *how* something may be implemented. Sometimes refinement is referred to as stepwise development enabling correctness-by-construction.

Two types of refinement exists, vertical known as data refinement [10] and horizontal known as superposition refinement [20] [11]. Coarsely, data refinement reduces non-determinism, strengthens the guard or defines more precise variables requiring a gluing invariant whilst superposition refinement introduces new variables and events. The refinement relations are associated with conditions that need to be proved. Moreover, refinements are monotone. This is central, i.e. assume components C and M that make up specification $C \parallel M$, then as of monotony if $M \sqsubseteq M'$ then trivially $C \parallel M \sqsubseteq C \parallel M'$.

3. The In-House Temperature Control System Specification

In this section we will report on the stepwise development of an in-house temperature control system (*tcs*). The *tcs* sole purpose is to maintain the in-house temperature within a predefined interval. This involves a sensor sensing the temperature as well as actuators affecting the temperature in a stigmergic manner, in this case the heating and cooling unit(s). The stepwise development is realised by refining features to a more abstract specification. Figure 2 depicts this refinement process on general level showing the refinement relations by arrows from the initial machine **M0** to the most concrete machine **M4**. In writing, the specifications' comments are delimited by //.

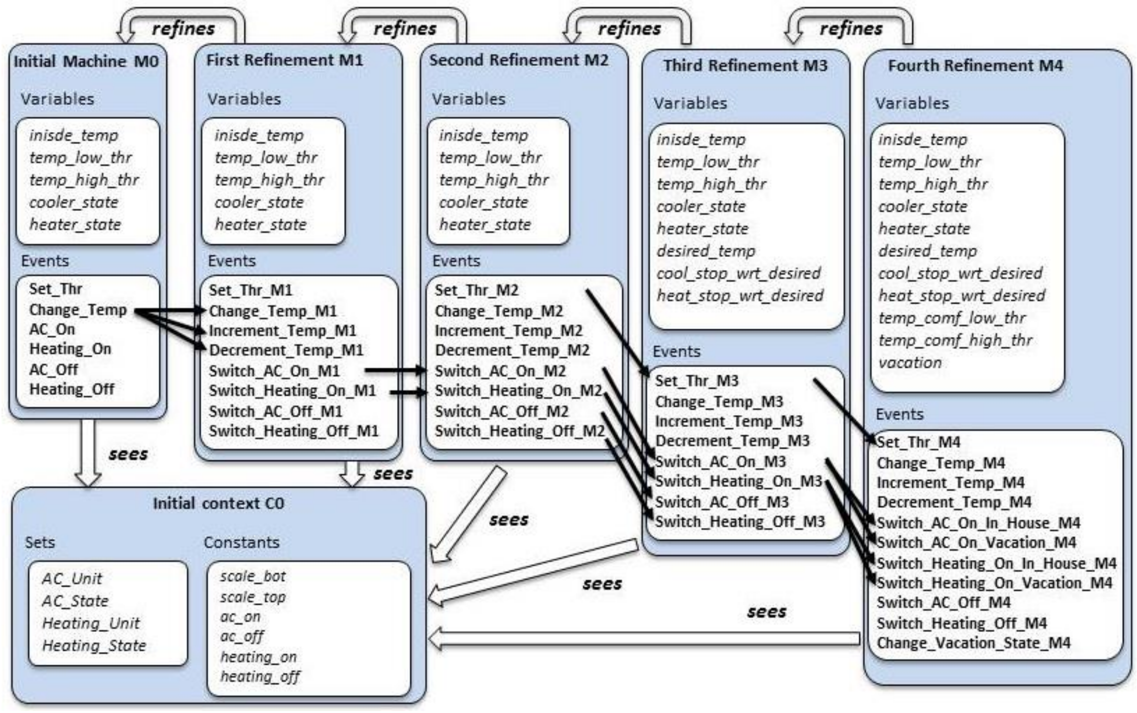

Figure 2: The *tcs* stepwise development process

3.1. The Initial Specification M0 and Context C0

The initial machine **M0** of *tcs* defines the operational environment by seven events. For the initial machine, obvious invariants of the system include those defining the types of the variables. These include cooler_state ∈ **AC_Unit** → **AC_State** and heater_state ∈

Heating_Unit → Heating_State declaring that each unit is assigned a state. The units are defined in the context **C0** that machine **M0** sees.

In addition to the sets, the context **C0** declares scale_top and scale_bottom. Realistically these could be considered in °C and with the assumption that no realistic temperature in-house should be outside this interval. This assumption implies that, for example, a sauna is not considered. Moreover, the context declares that the AC_State is always either ac_on or ac_off; and likewise for the Heating_State on heating_on and heating_off.

```
context C0
constants   scale_top
            scale_bottom
            ac_on
            ac_off
            heating_on
            heating_off

sets        AC_Unit
            AC_State
            Heating_Unit
            Heating_State

axioms
   @axm1  scale_top = 100
   @axm2  scale_bottom = −100
   @axm3  partition(AC_State, {ac_on}, {ac_off})
   @axm4  partition(Heating_State, {heating_on}, {heating_off})
end
```

For the initial machine **M0**, the invariants @inv1 to @inv3 merely declares the type of the variables, here as an integer $\mathbb{Z}$. The invariant @inv4 defines the relation on temperatures and their thresholds. Notable for @inv4 is that it does not regard the variable inside_temperature to be bound within the low and high thresholds. This is obviously a feature later refined into the specification. Invariants @inv5 defines variables cooler_state = {ac_on, ac_off} as a function on an AC_Unit to AC_State, and similarly for heater_state in @inv6 on Heating_Unit and Heating_State.

For the events, the **INITIALISATION** event assigns the variables their initial values where the most notable is that all actuator units are set to the off state. The inside_temperature is assigned any value within scale_bottom and scale_top. For the remaining event, **Set_Threshold_M0** narrows the allowed temperature by assigning temperature_low_threshold any value less than inside_temperature - 1 and temperature_high_threshold any value over inside_temperature + 1. A feature of **Set_Threshold_M0** is that the thresholds may be changed dynamically.

Event **Change_Temperature_M0** update the temperature non-deterministically with 1°C granularity in either direction. Notable is that at this level we have not yet specified any restrictions on inside_temperature. The remaining four events **Switch_AC_on_M0**, **Switch_AC_off_M0**, **Switch_Heating_on_M0** and **Switch_Heating_off_M0** specify switching the state of an AC_Unit or Heating_Unit. Their respective guards ensure that a unit may be switched off only if there is one that is on, and vice versa.

```
machine M0 sees C0
variables   inside_temperature // inside temperature
            temperature_low_threshold // low threshold for acceptable temperature
            temperature_high_threshold // high threshold for acceptable temperature
            cooler_state
            heater_state

invariants
   @inv1    inside_temperature ∈ ℤ
   @inv2    temperature_low_threshold ∈ ℤ
   @inv3    temperature_high_threshold ∈ ℤ
   @inv4    scale_bottom ≤ temperature_low_threshold ∧ temperature_low_threshold ≤
               temperature_high_threshold ∧ temperature_high_threshold ≤ scale_top
   @inv5    cooler_state ∈ AC_Unit → AC_State
   @inv6    heater_state ∈ Heating_Unit → Heating_State

events
 event INITIALISATION
  then
   @act1    inside_temperature :∈ scale_bottom‥scale_top
   @act2    temperature_low_threshold ≔ scale_bottom
   @act3    temperature_high_threshold ≔ scale_top
   @act4    cooler_state ≔ AC_Unit × {ac_off} // assign all AC_units to off
   @act5    heater_state ≔ Heating_Unit × {heating_off} // assign all heating units to off
 end

 event Set_Thresholds_M0
  any       low high
  where
   @grd0_1  low ∈ scale_bottom‥inside_temperature − 1 // defines granularity to 1
   @grd0_2  high ∈ inside_temperature + 1 ‥scale_top // at least 2 degrees difference
  then
   @act0_1  temperature_low_threshold ≔ low
   @act0_2  temperature_high_threshold ≔ high
 end

 event Change_Temperature_M0
  any       change
  where
   @grd0_1  change = −1 ∨ change = 1
  then
   @act0_1  inside_temperature ≔ inside_temperature + change
 end

 event Switch_AC_On_M0
  any       ac_cooling_unit
  where
   @grd0_1  ac_cooling_unit ∈ AC_Unit ∧ cooler_state(ac_cooling_unit) = ac_off
  then
   @act0_1 cooler_state(ac_cooling_unit) ≔ ac_on
 end

 event Switch_Heating_On_M0
  any       heat_unit
```

```
    where
      @grd0_1  heat_unit ∈ Heating_Unit ∧ heater_state(heat_unit) = heating_off
    then
      @act0_1  heater_state(heat_unit ≔ heating_on
  end

  event Switch_AC_Off_M0
    any       ac_cooling_unit
    where
      @grd0_1  ac_cooling_unit ∈ AC_Unit ∧ cooler_state(ac_cooling_unit) = ac_on
    then
      @act0_1  cooler_state(ac_cooling_unit) ≔ ac_off
  end

  event Switch_Heating_Off_M0
    any       heat_unit
    where
      @grd0_1  heat_unit ∈ Heating_Unit ∧ heater_state(heat_unit) = heating_on
    then
      @act0_1  heater_state(heat_unit) ≔ heating_off
end
```

3.2. The First Refined Machine M1

Machine **M1** refines **M0**, i.e. **M0** ⊑ **M1**. More specifically, **M1** specifies how the temperature may change, i.e. that inside_temperature > temperature_low_threshold and inside_temperature < temperature_high_threshold as by in @inv7. This is realised by refining **Change_Temperature_M0** to three events, **Increment_Temperature_M1**, **Decrement_Temperature_M1** and **Change_Temperature_M1**. The new events assure by guards that only one of them may be enabled at any given time as well as that if an actuator is on, then the temperature may only change accordingly without breaching the thresholds, i.e. new guards @grd1_1, @grd1_2 and @grd1_3. Here notable is that @grd1_1 ⇒ @grd0_1, i.e. strengthens the specific event's guard being a refinement.

When all **AC_Unit** and **Heating_Unit** are off, **Change_Temperature_M1** guards @grd1_2_1 and @grd1_2_2 evaluate true and the temperature may change non-deterministically within the allowed interval, assured by @grd1_3_1 and @grd1_3_2. This is illustrated in Figure 3 with abbreviated variable names. For execution, the **INITIALISATION** may have assigned inside_temperature either scale_bottom or scale_top in case either the inside_temperature need to be increased or decreased by first switching an **AC_Unit** or **Heating_Unit** appropriately on. This is assured by @grd1_3 in **Increment_Temperature_M1** that is the only enabled temperature changing event when inside_temperature = scale_bottom; and dually for **Decrement_Temperature_M1** and inside_temperature = scale_top.

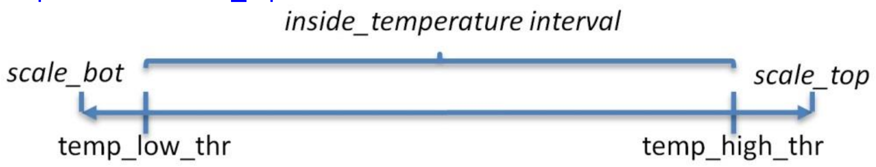

Figure 3: *tcm* temperature interval

In addition, if **Change_Temperature_M0** is enabled, then one of **Increment_Temperature_M1**, **Decrement_Temperature_M1** or **Change_Temperature_M1** is enabled that covers for possible restrictions on the exit condition. In the specification **M1** below, we omit non-changed variables, invariants and events.

```
machine M1 refines M0  sees C0
variables     // as in M0

invariants    // @inv1 through @inv6 as in M0
    @inv7     temperature_low_threshold ≤ inside_temperature ∧
                  inside_temperature ≤ temperature_high_threshold

events
… // INITIALISATION, Set_Thresholds_M1, Switch_AC_On_M1, Switch_AC_Off_M1,
Switch_Heating_Off_M1 as in their namesake events of M0
  event Increment_Temperature_M1
   refines Change_Temperature_M0
    any         change
    where
      @grd0_1     change = −1 ∨ change = 1
      @grd1_1     change = 1
      @grd1_2     ∃x ·x ∈ Heating_Unit ∧ heater_state(x) = heating_on // A Heating_Unit is on
      @grd1_3     inside_temperature < temperature_high_threshold
    then
      @act0_1     inside_temperature ≔ inside_temperature + change
  end

  event Decrement_Temperature_M1
   refines Change_Temperature_M0
    any         change
    where
      @grd0_1     change = −1 ∨ change = 1
      @grd1_1     change = −1
      @grd1_2     ∃x · x ∈ AC_Unit ∧ cooler_state(x) = ac_on // An AC_Unit is on
      @grd1_3     inside_temperature > temperature_low_threshold
    then
      @act0_1     inside_temperature ≔ inside_temperature + change
  end

  event Change_Temperature_M1
   refines Change_Temperature_M0
    any         change
    where
      @grd0_1     change = −1 ∨ change = 1
      @grd1_2_1 ∀x ·x ∈ AC_Unit ⇒ cooler_state(x) = ac_off
      @grd1_2_2 ∀x ·x ∈ Heating_Unit ⇒ heater_state(x) = heating_off
      @grd1_3_1 inside_temperature > temperature_low_threshold
      @grd1_3_2 inside_temperature < temperature_high_threshold
    then
      @act0_1     inside_temperature ≔ inside_temperature + change
  end
end
```

3.3. The Second Refined Machine M2

Machine **M2** refines **M1**, i.e. **M1** ⊑ **M2** and as of monotonicity **M0** ⊑ **M2** as well as **C0** || **M1** ⊑ **C0** || **M2**. Machine **M2** merely adds conditions for the functionality of switching an actuator on. Hence, the events of **M2** extend those of **M1**.

This switching is defined to happen in event **Switch_AC_On_M2** for an AC_Unit when temperature_high_threshold ≤ inside_temperature having the affect to starting cooling when the upper temperature bound is reached. The specification for Heating_Unit when temperature_low_threshold ≥ inside_temperature is a dual to **Switch_Heating_On_M2**. For the layout below, we omit the parts that remain unchanged.

```
machine M2 refines M1 sees C0
variables    // as in M1
invariants   // as in M1

events
… // INITIALISATION, Set_Thresholds_M2, Increment_Temperature_M2,
Decrement_Temperature_M2, Change_Temperature_M2, Switch_AC_Off_M2,
Switch_Heating_Off_M2 as in their namesake events of M1

 event Switch_AC_On_M2
  extends Switch_AC_On_M1
  where
   @grd2_1 temperature_high_threshold ≤ inside_temperature
 end

 event Switch_Heating_On_M2
  extends Switch_Heating_On_M1
  where
   @grd2_1 temperature_low_threshold ≥ inside_temperature
 end
end
```

3.4. The Third Refined Machine M3

Machine **M3** refines **M2**. Machine **M3** is the specification introducing adaptability. On this, **M3** introduces a delay in switching the actuators off. This is realised by new variables cool_stop_wrt_target and heat_stop_wrt_target that are relative to the new variable target_temperature. The motivation is the realistic situation that a heating unit emits heat for some time after switched off and dually a cooling unit cools with a delay; possible energy-wise penalties are discarded from consideration as this is an optimisation issue. Realistically, consider the variable target_temperature to be set by the inhabitant of the modelled space, however, in a manner never compromising the specification, i.e. considering the temperature changes instantaneous, then target_temperature could be set by 1° ticks. This temperature setting is depicted in Figure 4 with abbreviated variable names.

The invariants of **M3** guarantees that cooling and heating respectively must stop within the temperature interval, guaranteed by @inv9, @inv12 and @inv13. Initialisation event is obviously extended by the new variables. For the events providing functionality, event **Set_Thresholds_M3** extends **Set_Thresholds_M2** by introducing assignment of cool_stop_wrt_target and heat_stop_wrt_target as their relative difference from target_temperature.

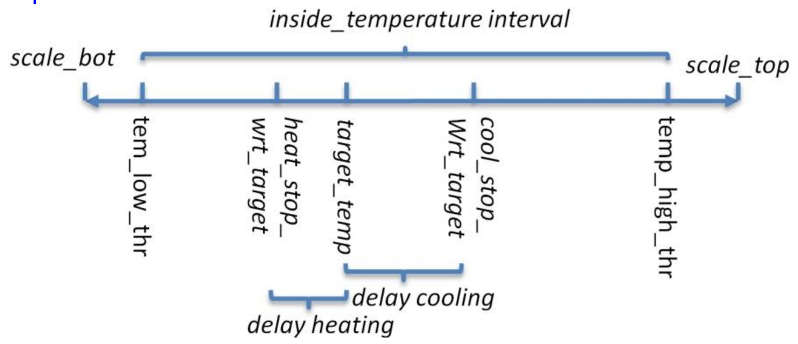

Figure 4: *tcm* with target temperature and delays

For the actuators' non-interfering functionality only the guards are strengthened. Hence, the events **Switch_AC_On_M3** and **Switch_Heating_On_M3** extend their abstract events by guards @grd3_1 assuring that there is no actuator of the other kind switched on. The effect of this is that no heating unit may be on simultaneously with a cooling unit, and vice versa. This is assured by invariants @inv14 and @inv15. For events **Switch_AC_Off_M3** and **Switch_Heating_Off_M3** the delay is taken into account by guards @grd3_1 of the respective event. This assures that the actuator may be switched off before reaching the target temperature. Again, in the outline below we omit non-changed parts of the specification.

```
machine M3 refines M2 sees C0
variables     … // variables as before
              target_temperature // the target temperature
              cool_stop_wrt_target // delay in cooling
              heat_stop_wrt_target // delay in heating

invariants    … // invariants as before
  @inv8       target_temperature ∈ ℤ // temperature to which the thermostat is set
  @inv9       temperature_low_threshold ≤ target_temperature ∧
              target_temperature ≤ temperature_high_threshold
  @inv10      cool_stop_wrt_target ∈ ℤ
  @inv11      heat_stop_wrt_target ∈ ℤ
  @inv12      cool_stop_wrt_target + target_temperature ≥ temperature_low_threshold
  @inv13      heat_stop_wrt_target + target_temperature ≤ temperature_high_threshold
  @inv14      (∃x · x ∈ AC_Unit ∧ cooler_state(x) = ac_on) ⇒
              (∀y · y ∈ Heating_Unit ⇒ heater_state(y) = heating_off)
  @inv15      (∃x · x ∈ Heating_Unit ∧ heater_state(x) = heating_on) ⇒
              (∀y · y ∈ AC_Unit ⇒ cooler_state(y) = ac_off)

events
… // Increment_Temperature_M3, Decrement_Temperature_M3, Change_Temperature_M3 as
in their namesake events of M2

 event INITIALISATION
  extends INITIALISATION
   then
    @act3_1   cool_stop_wrt_target ≔ 0
    @act3_2   heat_stop_wrt_target ≔ 0
    @act3_3  target_temperature :∈ scale_bottom‥scale_top
  end

  event Set_Thresholds_M3
    extends Set_Thresholds_M2
    any       target c_stop h_stop
    where
      @grd3_1  c_stop ∈ target‥high
      @grd3_2  h_stop ∈ low‥target
      @grd3_3  target ∈ low‥high
    then
      @act3_1  cool_stop_wrt_target ≔ c_stop − target
      @act3_2  heat_stop_wrt_target ≔ h_stop − target
      @act3_3  target_temperature ≔ target
  end

  event Switch_AC_On_M3
    extends Switch_AC_On_M2
    where
      @grd3_1 ∀x · x ∈ Heating_Unit ⇒ heater_state(x) = heating_off
  end

  event Switch_Heating_On_M3
    extends Switch_Heating_On_M2
    where
      @grd3_1 ∀x · x ∈ AC_Unit ⇒ cooler_state(x) = ac_off
  end

  event Switch_AC_Off_M3
    extends Switch_AC_Off_M2
    where
      @grd3_1 inside_temperature + cool_stop_wrt_target ≤ target_temperature
  end

  event Switch_Heating_Off_M3
    extends Switch_Heating_Off_M2
    where
      @grd3_1 inside_temperature + heat_stop_wrt_target ≥ target_temperature
  end
end
```

3.5. The Fourth Refined Machine M4

Machine **M4** refines **M3**. It introduces the more restrictive temperature threshold providing a sense of comfortability by defining variables vacation and new thresholds temperature_comfortable_low_threshold and temperature_comfortable_high_threshold. The vacation is false when the inhabitants of the controlled house are in-house and true otherwise. Hence, when vacation is true, the less restrictive temperature interval as specified in the more abstract machines is applied. Dually, when vacation is false, the new thresholds providing a sense of comfortability introduced in this machine are applied. The temperature thresholds and intervals are illustrated in Figure 5.

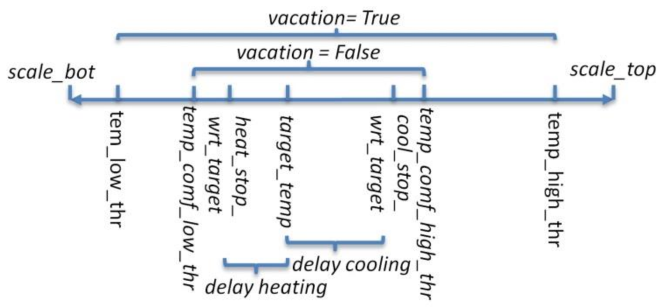

Figure 5: tcm with delay and vacation

The ordering of the new thresholds and temperature levels are declared by invariants, in this case @inv18 through @inv22. This is realised by events **Set_Thresholds_M4** that extends **Set_Thresholds_M4** by @grd4_1 with @act4_1 and @grd4_2 with @act4_2 that assign temperature_comfortable_low_threshold a value in interval temperature_low_threshold .. target_temperature and similarly for temperature_comfortable_high_threshold a value in interval target_temperature .. temperature_low_threshold. Moreover, the switching on of the actuators are extended to include vacation, i.e. actuators are switched on at temperatures depending on the state of vacation. This is implemented by straightforward strengthening of the guards. In addition, an event **Change_Vacation_State_M4** capturing switching the vacation state is specified. Again we omit non-changed parts of the specification.

```
machine M4 refines M3 sees C0
variables    … // variables as before
             temperature_comfortable_low_threshold
             temperature_comfortable_high_threshold
             vacation

invariants   … // invariants as before
  @inv16     temperature_comfortable_low_threshold ∈ ℤ
  @inv17     temperature_comfortable_high_threshold ∈ ℤ
  @inv18     temperature_comfortable_low_threshold ≤
                 temperature_comfortable_high_threshold
  @inv19     temperature_low_threshold ≤ temperature_comfortable_low_threshold ∧
                 temperature_comfortable_low_threshold ≤
                 cool_stop_wrt_target + target_temperature
  @inv20     temperature_comfortable_high_threshold ≤ temperature_high_threshold ∧
                 heat_stop_wrt_target + target_temperature ≤
                 temperature_comfortable_high_threshold
  @inv21     cool_stop_wrt_target + target_temperature ≥
                 temperature_comfortable_low_threshold // inv 21 strengthens inv 12
  @inv22     heat_stop_wrt_target + target_temperature ≤
                 temperature_comfortable_high_threshold  // inv 22 strengthens inv 13
  @inv23     vacation ∈ BOOL

events
… // Increment_Temperature_M4, Decrement_Temperature_M4, Change_Temperature_M4,
Switch_AC_Off_M4, Switch_Heating_Off_M4  as in their namesake events of M3

 event INITIALISATION
  extends INITIALISATION
  then
    @act4_1 temperature_comfortable_low_threshold ≔ scale_bottom
    @act4_2 temperature_comfortable_high_threshold ≔ scale_top
    @act4_3 vacation :∈ BOOL
 end

 event Set_Thresholds_M4
  extends Set_Thresholds_M3
  any l_temp h_temp
  where
    @grd4_1 l_temp ∈ low‥target
    @grd4_2 h_temp ∈ target‥high
  then
    @act4_1 temperature_comfortable_low_threshold ≔ l_temp
    @act4_2 temperature_comfortable_high_threshold ≔ h_temp
 end

 event Switch_AC_On_In_House_M4
  extends Switch_AC_On_M3
  where
    @grd4_1 vacation = FALSE
    @grd4_2 temperature_comfortable_high_threshold ≤ inside_temperature
 end

 event Switch_AC_On_Vacation_M4
  extends Switch_AC_On_M3
  where
    @grd4_1 vacation = TRUE
 end

 event Switch_Heating_On_In_House_M4
  extends Switch_Heating_On_M3
  where
    @grd4_1 vacation = FALSE
    @grd4_2 temperature_comfortable_low_threshold ≥ inside_temperature
 end

 event Switch_Heating_On_Vacation_M4
  extends Switch_Heating_On_M3
  where
    @grd4_1 vacation = TRUE
 end

 event Change_Vacation_State_M4
  then
    @act4_1 vacation :∈ BOOL
 end
end
```

4. Discussion

This paper is a report on stepwise specification of an in-house *tcs*. The specification was developed in a stepwise manner utilising the refinement of Event-B specification. Within the reporting of this development, we have tried to point out how this refinement is realised. Not surprisingly, we had a "refinement strategy" in mind when starting to specify. The specified *tcs* employ a flexible temperature interval. Moreover, it adapts to the possibly less restrictive requirements of its inhabitants, i.e. when they are on vacation. Obviously, this indication of vacation could be used for other purposes as well, such as for burglar alarm or proactively switching lights on and off as if the house was inhabited.

The correctness of the specification relies on some general assumptions with respect to the modelled environment. This is a reasonable consequence of seeking mathematical correctness in the first place, where it is obvious that without assumptions on the environment making the model precise, no mathematical correctness could have been shown. Hence, as mentioned in the introduction, the specification is correct on an environment that adheres to the model; whose distance from the "real" environment is out of the scope of this paper and thus not considered. However, we acknowledge that these assumptions simplify the imperfect matters that are causes of faults and failures [25].

With this, we allow us to assume in **M1** that the power of any **AC_Unit** and any **Heating_Unit** exceeds its countering power, i.e. that if the heating is turned on, the house will heat up. Realistically this means that we assume that if heating is on, the windows and doors are closed, or open to an extent whose countering effect is less than that of the actuator. Moreover, the specification assumes all the data to be correct, i.e. that the switch of an actuator is never broken and that the sensors always work. Hence, the machines specified in this paper outline how a control system would work in an idealised environment lacking any kind of erroneous or inconsequent behaviour, those captured by the model. Other assumptions characterising to the specification outlined in this report includes the granularity of temperature that is set to 1°C in **M0**. On this we wish to note that the outcome would be identical if we would have added decimals.

5. Conclusion

Means to manage complexity and assurance of behaviour are the most important reasons for formally specifying a system. As the *tcs* specified in this report is rather simple and straightforward, the motivation is in the possibility to integrate this as part of some greater control system. Examples of interesting systems by which integration could be interesting include that of lights control system featuring motion sensors [27]. Together, *tcs* with variable vacation and lights control system motion sensors could cover for a burglar detection system. In such cases, the formally specified boundaries, variable names and state space of a subsystem are of outmost importance. Formal support for this is briefly introduced in Section 2.2 overviewing the capabilities related to (de)composition of a formal specification.

In this paper we have reported on the stepwise development of one part of an adaptive house's control systems, the *tcs*. The development is done independently of any other but aims for being integrated with other control systems. We claim that if an implementation would be engineered based on this specification, it would be correct with respect to the assumptions on the model. Moreover, we claim that the integration would be formally provable by parallel composition. As for the proof of the specifications and refinements presented in this paper, we note that the Rodin Platform tool's theorem provers did automatically discharge all proof obligations.

References

[1] J.-R. Abrial, "Event Model Decomposition," Wiki document Version 1.3 http://wiki.event-b.org/images/Event_Model_Decomposition-1.3.pdf accessed 15 May 2013, 2009.

[2] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering.*: Cambridge University Press, 2010.

[3] J.-R. Abrial, *The B-Book: Assigning programs to meanings*. New York, USA: Cambridge University Press, 1996.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, and F. and Voisin, L. Mehta, "Rodin: An Open Toolset for Modelling and Reasoning in Event-B," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, no. 6, pp. 447-466, 2010.

[5] J.-R. Abrial et al., "Rodin: An Open Toolset for Modelling and Reasoning in Event-B.," *International Journal on Software Tools forTechnology Transfer (STTT)*, vol. 6, pp. 447-466, 2010.

[6] J.-R. Abrial, S. Schuman, and B. Meyer, "A Specification Language," in *On the Construction of Programs.*: Cambridge University Press, 1980.

[7] F. Arbab, "Reo: A Channel-based Coordination Model for Component Composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329-366, 2004.

[8] R. Back, "On the correctness of refinement steps in program development," Department of Computer Science, University of Helsinki, PhD thesis Report A-1978-4 1978., 1978.

[9] R. Back and R. Kurki-Suonio, "Decentralization of Process Nets with Centralized Control," in *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1983.

[10] R. Back and K. Sere, "Stepwise Refinement of Action Systems," *Structured Programming*, vol. 12, no. 1, pp. 17-30, 1991.

[11] R. Back and K. Sere, "Superposition Refinement of Reactive Systems," *Formal Asp. Comput.*, vol. 8, no. 3, pp. 324-346, 1996.

[12] R. Back and J. von Wright, *Refinement Calculus: a Systematic Introduction.*: Springer-Verlag New York, 1998.

[13] M. Butler, "Decomposition Structures for Event-B," in *Integrated*

Formal Methods iFM2009, 2009.

[14] (2013, May) Decomposition Plug-in User Guide. [Online]. http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide

[15] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," vol. 18, 8 , no. Commun. ACM, pp. 453-457, 1975.

[16] E. Emerson and J. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *Journal of Computer and System Sciences* , vol. 30, no. 1, pp. 1–24., 1985.

[17] (2013, May) Event-B and the Rodin Platform. [Online]. http://www.event-b.org/

[18] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576 - 580, 583, October 1969.

[19] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[20] S. Katz, "A superimposition control construct for distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, pp. 337-356, 1993.

[21] K. Mani Chandy, *Parallel Program Design: a Foundation*.: Addison-Wesley Longman Publishing Co., Inc., 1988.

[22] R. Milner, *A Calculus of Communicating Systems*.: Springer Verlag, 1980.

[23] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Inf. Comput.* , vol. 100, no. 1, pp. 1-40, 1992.

[24] C. Morgan, *Programming from Specifications*. Upper Saddle River, NJ, USA: Prentice-Hall Inc., 1990.

[25] D. Parnas, "Really Rethinking 'Formal Methods," *Computer*, vol. 43, no. 1, pp. 28-34, 2010.

[26] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46-57.

[27] P. Sandvik, "Formal Stepwise Development of an In-House Lighting Control System," TUCS Technical Reports nr. 1079 2013.

[28] K. Sere, "Stepwise derivation of parallel algorithms," Åbo Akademi, Department of computer science, PhD Thesis ISBN: 951-649-748-9, 1990.

ISBN 978-952-12-2893-3
ISSN 1239-1891