Basic Research in Computer Science

Danfoss EKC Trial Project Deliverables

Kim G. Larsen
Ulrik Larsen
Brian Nielsen
Arne Skou
Andrzej Wasowski

BRICS Report Series **RS-03-48**

ISSN 0909-0878 **December 2003**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/48/`

Danfoss EKC Trial Project Deliverables

Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, Andrzej Wasowski
Department of Computer Science
Aalborg University
Fredrik Bajersvej 7E
DK-9220 Aalborg, Denmark
{kgl,ulrikl,bnielsen,ask,wasowski}@cs.auc.dk

December, 2003

Abstract

This report documents the results of the Danfoss EKC trial project on model based development using IAR visualState. We present a formal state-model of an refrigeration controller based on a specification given by Danfoss. We report results on modeling, verification, simulation, and code-generation. It is found that the IAR visualState is a promising tool for this application domain, but that improvements must be done to code-generation and automatic test generation.

Contents

1 Introduction

This paper documents the results of a trial project on model based development of embedded systems in collaboration with Danfoss "Refrigeration and A/C Controls Division" (DCD).

1.1 Objectives

The objective of the study is to quickly form a basis on which a decision can made about whether model based development using formal UML notation and an appropriate tool for simulation, verification, test and code generation appear to be feasible in the specific application domain of DCD low-end products, that phase one of the planned collaboration project between CISS and Danfoss should be initiated.

To meet this objective it was agreed that DCD provides a small but realistic application that CISS attempts to model using a state-of-the-art tool, IAR visualSTATE®. The application provided by Danfoss is a typical sub-function of an EKC-thermostat—an industrial refrigeration and air-conditioning controller. The result should at least make it plausible that DCD application software can be realized using IAR visualSTATE®, that this can be integrated with the input/output parameter structure of the EKC, and that the model can be used as basis for coding and testing.

The deliverables consists of

- This report documenting and summarizing the trial project.
- The developed IAR visualSTATE® models.
- A developed PC based demo application.
- An oral presentation of the result to DCD representatives.

1.2 The process

The modeling has been done during a period of approximately one month, including about 2 weeks of Christmas and New Year break. The work has progressed gradually and iteratively towards the finalized model. The following main steps have been identified:

1. **Study of application requirements:** The EKC-Thermostat specification was studied in detail to understand the required functionality. During this activity and the initial modeling in the next step, several questions were raised about the required functionality.
2. **Model v1 (Doodling)** The first draft of the model was developed by doodling on a whiteboard. Also options about what events the environment should gen-

erate and how the model should be integrated with the DCD device drivers and parameter database were discussed.

3. **Model v2 (Flat model)**. The first formalized model was entered into IAR visualSTATE® and syntactically checked. The aim was to identify main concurrent components (temperature calculator, regulator, alarm handler) and modes (normal, emergency, defrost and standby).

4. **Model v3 (Hierarchical model)**. After the model v2 followed a series of simplifications/refinements and resolutions of the specification ambiguities. This was mainly done by refactoring redundant information by utilizing hierarchical state machines, entry- and -exit transitions, and internal reactions.

5. **Verification and Code Generation** As model v3 progressed more time was spent on checking the functionality. This was done through reviewing and discussing the model, running it through the standard verification checks of IAR visualSTATE®, and by animating it using IAR visualSTATE® and a small PC based demo application generated by IAR visualSTATE®.

1.3 Staff

The following CISS staff members contributed to the modeling:

Kim Larsen: Full Professor. super expert on general state machine modeling and verification algorithms, has conducted several industrial case studies. Time spent: 1 day

Ulrik Larsen: New research assistant. IAR visualSTATE® tool developer, contacts with the IAR visualSTATE® company. Time spent: less than 1 day.

Brian Nielsen: Associate Professor. Expert on model based testing, some industrial case studies using model based testing. Main responsible for editing the model using IAR visualSTATE®; no prior detailed knowledge about IAR visualSTATE® or UML, and no previous application of IAR visualSTATE®. Time spent: 5 days.

Arne Skou: Associate Professor. Modeling expert, several industrial case studies on modeling and verification. Time spent: 1 day.

Andrzej Wasowski Senior Phd Student. Expert on IAR visualSTATE® and UML modeling language and semantics, and code generation expert. Main responsible for code generation and demo application development. Time spent: 3 days.

2 Modeling the EKC

2.1 Environment Integration

The model is based on a fundamental choice regarding its interaction with the DCD parameter database and i/o drivers, that constitute the environment of the model. The current DCD software is divided into two main parts, the parameter database and i/o drivers, and the application logic. These parts communicate through a set of shared global variables classified as either input variables, output variables, or settings variables (special user input variables containing dynamically configurable parameters). The parameter interface is depicted in Figure 1.

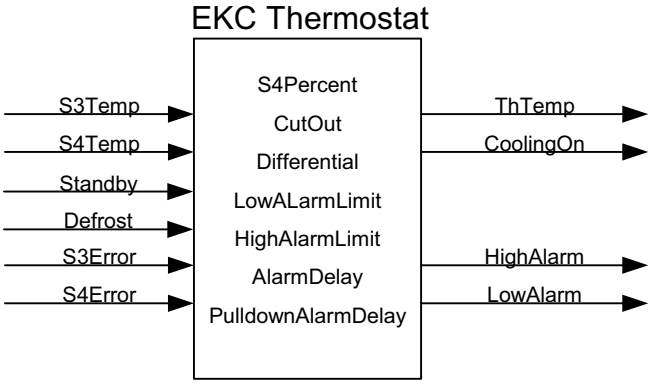


Figure 1: Parameter Interface

The code generated by IAR visualSTATE® requires that events are communicated to the state machine interpreter when the environments changes such that the machine is required to react. This is done by writing a small software adaptation layer relaying model input/output events to and from the device driver interface. It is required by DCD that the current parameter database is reused. This can be done in two principally different ways:

Pure Event based: In collaboration with the i/o drivers and parameter database the adaptation software generates events (like standbyRequested, StandbyDisabled, temperatureSample, s3ErrorDetected, S3ErrorRepaired etc).

Pure Shared Variable Based: The model is requested to read and write the shared variables periodically or at requested points in time, and then react accordingly.

The shared variable based approach was selected due to the natural and easy integration with DCD interface. *The model assumes that a set of globally shared variables exists named according to the DCD specification (See Figure 1), and that the model is informed via a single* **sample** *event by the adaptation software when it is required to react to the current values of the variables*. It is also assumed that the model can order

timer signals from the underlying platform, and that model can be informed about these via adaptation generated events.

It also turns out to be *convenient* to assume that the environment generates **sample** events whenever one of the parameters change. In particular it is assumed that each change to the standby and defrost mode variables are reported via separate **sample** events, i.e if (standbyOn==1, defrostActive==1) changes to (standbyOn==0, defrostActive==0) this is reported via 2 **sample** events revealing the intermediate configuration.

2.2 Walk Through of the Finalized model

The final model is depicted in Figure 2. The top level state *TC* is divided into three concurrent regions: the *Regulator*, the *HighAlarmHandler*, and the *LowAlarmHandler*. The internal reaction rules of *TC* ensures that a new combined temperature estimate from the temperature sensors S3 and S4 is computed at each sample event, and that the other components reacts to this by issuing the *NewTemp* signal.

The *Regulator*

The *Regulator* contains a hierarchical state machine whose overall responsibility is to regulate the cooler, i.e., whether cooling is on or off, and for how long. Overall, the *Regulator* can be either in operation or in standby mode represented by the composite states *InOperation* and *StandingBy*. We assume that active regulation must be off when standing by.

When the system is *InOperation*, it can either regulate (*Regulating*) the temperature or or perform defrost operations (*DefrostMode*). The requirements specification does not describe what activity is performed in defrost mode.

The temperature regulation in *Regulating* state is different depending on whether there are sensor errors (*EmergencyMode*) or not (*NormalMode*). In *NormalMode* regulation is based on the calculated temperature estimate. The entry and exit rules of the substates *CoolerOff* and *coolerOn* accumulates the amount of time spent in each mode, and cuts-in or cuts-out cooling. Cooling is cut-in when the calculated temperature *ThTemp* exceeds the threshold parameter (*ThCutOut*) plus a hysteresis value (*thDifferential*) and cut-out when dropping below the threshold value *ThCutOut*.

EmergencyMode is entered when both temperature sensors fail. Depending on whether there is a history of normal operation (counted by the number of cut-outs (*CutOutCount*)) or not, the required cooling on time (*onTime*) in each 20 minute period in emergency mode is computed from the history or from a default 30/70% distribution, as required in specification section 1.1.1 and 1.1.2, hence the two transition.

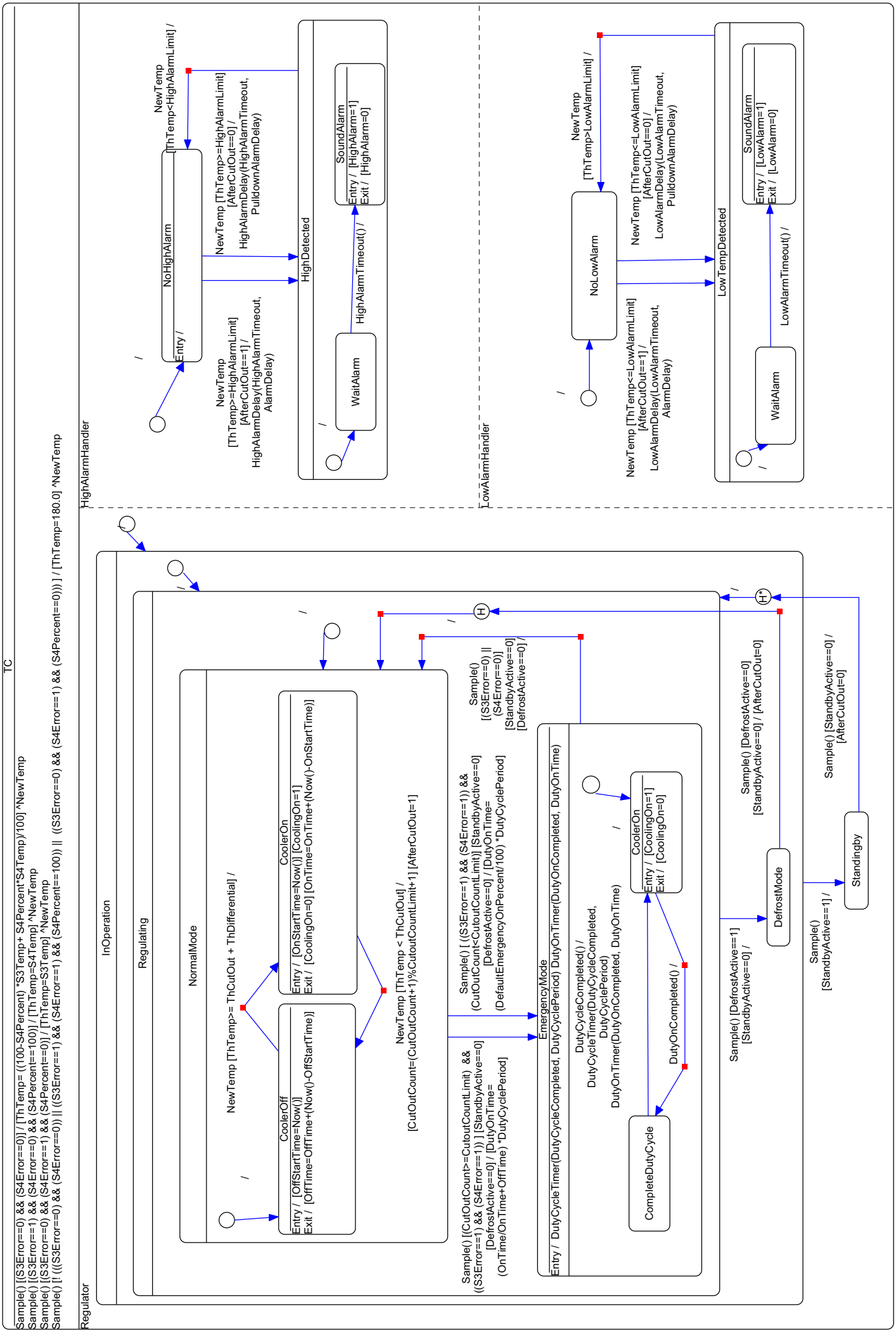


Figure 2: EKC state machine model

Two timers controls the time spent in the the sub-states. The *(*DutyOnTimer) issues event *DutyOnCompleted* when the required *onTime* has elapsed. Similarly, the *(*DutyCycleTimer) issues event *DutyCycleCompleted* when the required *DutyCyclePeriod* of 20 minutes has elapsed. The timer events are ordered whenever the *EmergencyMode* is entered. Emergency mode is exited as soon as one sensor becomes valid.

The *HighAlarmHandler*

The *HighAlarmHandler* enters state *HighDetected* when a calculated temperature *ThTemp* exceeds the threshold value for high temperature alarms (*HighAlarmLimit*). Depending on whether the threshold was reached before the first cut out after power on, leaving defrost or standby mode (as tracked by the boolean variable *AfterCutOut*), the delay before raising the alarm is respectively *AlarmDelay* or *PullDownAlarmDelay*. This delay between waiting for an alarm (*WaitAlarm*) and sounding it (*SoundAlarm*) is controlled by the timer *HighAlarmDelay* that generates the event *HighAlarmTimeout* upon expiration. The *HighDetected* state is left when the temperature is below the alarm limit whether the alarm as sounded or not (i.e the alarm may be canceled).

The *LowAlarmHandler*

The *LowAlarmHandler* is similar to the*HighAlarmHandler*.

2.3 Questions Raised during Modeling

During this activity and the initial modeling in the next step, several questions were raised about the required functionality. Essentially, these questions were caused by the informal requirements specification that have several unclear points and possible interpretations, and the fact that the staff are modeling experts, but lacks the specific knowledge of EKC-thermostat domain experts.

A few of the initial questions were directed to DCD, but most was resolved by the modelers using educated guesses of the intended behavior. This was done to facilitate a discussion later with DCD about how formal modeling can be used to identify possible ambiguities and design alternatives.

We apply the following terminology Q: question, A: answer, D: Design decision implied by the question/answer.

Q1 Should alarms be raised in standby and defrost modes?

A1 Finn Rasmussen, 11/12-03: Yes, but the pull-down timer delay should apply in these modes.

Q2 How are alarms cleared?

A2 Finn Rasmussen 11/12-03: Alarms are cleared immediately when their enabling conditions become false.

Q3 Are the cooling conditions to be calculated at a fixed frequency?

A3 Finn Rasmussen 11/12-03: This should be left open in the model, i.e. the model should not rely on any given frequency.

D3 CISS 18/12-03: We assume in the initial model that the database variables/parameters are sampled whenever the event 'sample' occurs. The necessary code to provide this event is part of the implementation adaptation.

Q4 Is the history of cut-in's (cut-out's) ever reset after power up?

Q5 Is there an upper limit for the frequency of cut-in/cut-out sequences?

E.g. fast switching between S3Error==true and S3Error==false might cause fast cut-in/cut-out switching.

Q6 Do all cut-in's contribute to the history statistics - e.g. when mode changes occur?

Q7 How must sensor errors be handled in non-operational modes (like e.g. defrost mode)?

Q8 Must decisions (e.g. cut-in decisions) be based on actual values - or is it acceptable to perform the cut-in during the next sample?

D8 CISS 18/12-03: This is a tool specific question related to the handling of signals. In the initial model we assume that actions are based on actual values.

Q9 There is an inconsistency on page 3 and page 5 leaving it unclear when to apply the pull-down delay (especially in the LowAlarmLimit case).

Q10 Is there an upper limit for the frequency of alarm settings?

Q11 Must sensor errors trigger any kind of alarms?

A11 Finn Rasmussen 11/12-03: In fact alarms should be raised, but we will ignore it in the trial case study.

Q12 When does a new period start in emergency mode (e.g. should a new period start even if emergency mode is only left for a short periods of time)?

Q13 What happens in when the system is requested to standby?

D13 System is required to disable cooling while in standby mode?

Q14 What is the possible transitions between standby and other modes, and what happens when standby is disabled?

D14 We assume that standby takes priority over other modes, and that standby preempts these (i.e. defrost, standby, and regulating are assumed to be mutually exclusive, and not concurrent. The interrupted mode is assumed to be resumed after standby.

Q15 It seems possible that both low alarm and high alarm may both sound at the same time, provided that the temperature varies sufficiently quickly.

D15 Ensured by the *concurrent* components handling of low alarm and high alarm respectively.

Q16 What are the exact conditions for detecting sensor error and using emergency cooling? Must S3 and S4 sensors both fail simultaneously, or only one of them? Does it depend on the settings of S4percent?

D16 Both sensors must fail.

Q17 What is the exact technical hack/solution for computing emergency onTime (running average) percentage based on cut/in and out history?

Q18 How is a duty cycle in emergency mode executed, i.e. how is the onTime distributed in the 20 minute cycle period?

A18 Finn Rasmussen 11/12-03???: Cooling is on in the beginning for the required time followed by cooling off for the remaining time.

2.4 Experiences

Our experiences shows that the IAR visualSTATE® modeling language is very expressive, and could be used to create an elegant, reasonable compact model that accurately reflects the behavior intended by the designers. A detailed understanding of the semantics of UML, and especially the IAR visualSTATE® dialect is necessary.

The graphical editor worked reliably, but requires some getting used to, and is a bit heavy to use due to many menus that must be opened. Some cut and paste operations (moving condition actions among transitions) could not be conveniently done within the graphical editor, and was performed directly in the textual format.

We also encountered a number of strange limitations:

- no expression allowed in timer value settings, but this must be precomputed variable.
- In boolean guard conditions it is not allowed to use variables of different types (so-called mixed mode expression).

- Range cannot be specified on external variables
- No default scoping rules for hierarchies, but prioritization must be done manually by adding additional conditions on transitions.

3 Verification results

The IAR visualSTATE® Verifier is a tool for formally and exhaustively checking the logical consistency of a IAR visualSTATE® model. The verifier automatically checks generic properties such as ambiguities (e.g. conflicting transitions or assignments), unactivated design elements, dead states etc.

3.1 Verification Example

During the development, the model have been checked with the verifier which has revealed a number of design problems that have been addressed in the later versions of the model. An example of an error report from an intermediate version of the model illustrated in Figure 3 is given below:

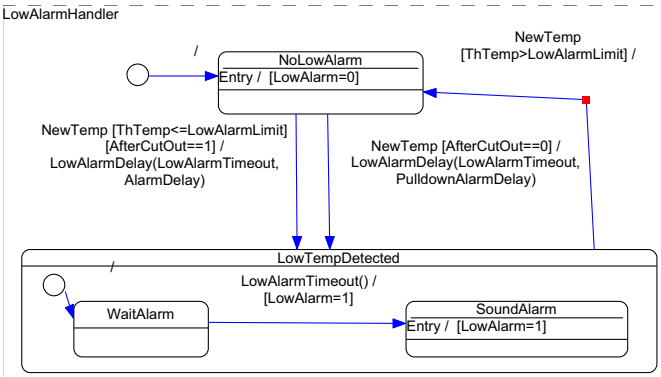


Figure 3: Illustration of conflict 2 in error report.

```
Ambiguous assignments (static check): (Error)
1) Error: The variable startTime is both assigned and read on the transition
CoolerOn:
 NewTemp [ ThTemp < ThCutOut ] / [CutOutCount=(CutOutCount+1)
                        %CutoutCountLimit+1] [AfterCutOut=1]
-> CoolerOff
CoolerOn:
 Exit / [OnTime=OnTime+(Now()-startTime)]
CoolerOff:
 Entry / [startTime=Now()] [CoolingOn=0]

2) Error: The variable LowAlarm is assigned several times on the transition
WaitAlarm:
```

```
 LowAlarmTimeout() / [LowAlarm=1]
-> SoundAlarm
SoundAlarm:
 Entry / [LowAlarm=1]
Fatal Error: Ambiguous system
Verification stopped
```

The reason for the ambiguous assignment is that *LowAlarm* is assigned twice upon entry to state *SoundAlarm*, once on the transition from *WaitAlarm* to *SoundAlarm*, and once on the entry action in state *SoundAlarm*. Although the values are identical in the example, and no harm is done, this was not intended.

3.2 Experiences

Unfortunately not all static checks could be completed due to an internal error in IAR visualSTATE®. Our experience shows that verification could be done on the model, but also that it required some care in choosing a usable verification mode and setting the verification parameters was a bit confusing and required some expertise (e.g.ṅumber of bits for variable encoding).

No application oriented user written properties was verified during the pilot project.

4 Model Simulation

The EKCThermostat model has been simulated using IAR visualSTATE® Validator.

4.1 The IAR visualSTATE® Validator

The simulation of the model is event driven. User provides the state of the environment (variables) and then fires an event. Validator advances the system state and gives information about the actions executed.

The EKC model has been specified using variable-interface, which makes it a bit tedious to simulate. In this specific case, in each step one has to modify variables in desired way and then send the Sample event. Simulator allows defining values for functions not available at so early stages. It also handles timers automatically (at selected speed) and visualizes the animation. The signal queue can be handled automatically, or single-stepped by the operator.

4.2 Experiences

Several function bugs were discovered in earlier versions of the EKC model during simulation in Validator. Most of them were caused by incomplete guards and some by misplaced actions.

In general the impression is that IAR visualSTATE® validator is a useful tool for functional testing at early design stages. It should be stressed that the simulation would have been easier if we had created a more natural, event-driven model. Also we have found that driving the model into some specific situations, deep in the hierarchy may be tedious, which indicates that "super-tracing" extension, one among many proposals in the project, would be a useful extension.

5 Code Generation

We have carried out experiments with code generation and compilation for two platforms (Renesas h8300 and ATMEL atmega16).

5.1 Size Results

The following table presents executable sizes[1] obtained with the H8 compiler. The IAR visualSTATE® (VS) column shows the results for the code generator supplied in the IAR visualSTATE® package. The SCOPE column shows the results for the experimental research-based code generator (SCOPE).

model	**IAR visualSTATE® 5.1** [bytes]	**SCOPE** [bytes]
minimal	1 748	1 312
aircond	2 426	1 668
EKC	4 134	est. 2 600

The aircond model is a small mockup of the airconditioner shown at the previous meeting. Minimal is a model containing two states and a single transition (no guard and action). EKC is the model we described earlier in the report. This model has been created in the newest version of IAR visualSTATE® Designer and cannot be currently compiled by SCOPE due to file format changes. The SCOPE result for EKC is estimated by proportional scaling of aircond size after substraction of the minimal model size. We expect to have SCOPE ported to the new format soon.

The EKC and the minimal model have also been compiled for ATMega16 platform, with the following results:

model	**IAR visualSTATE® 5.1** [bytes]
minimal	1 344
EKC	4 248

Then a precise analysis has been made on the EKC image produced for the ATMEL platform with the builtin IAR visualSTATE® coder, giving following results:

[1] Note that executable files contain more information than just the kernel with model data and code.

code/data description	**size** [bytes]
SemLibB+main loop (kernel)[2]	1 170
guards code	920
guards dispatcher (meta-code)	62
actions code	498
actions dispatcher (meta-code)	52
VS (transition table)	823

These numbers add up to a sum of about 3.5k. The remaining space (nearly 1k) is used by routines which are not needed and can be removed from linking in the production code and parts which will be needed anyway (for instance C startup code).

We used the same method (dissasemble with avr-objdump) to compute the size of current Danfoss Kernel, which had been made available to us. The kernel code, initialization and main loop occupies 580 bytes in this kernel, approximately half of the IAR visualSTATE® kernel size. We do not have any comparison of model representation size, as we do not yet have the EKC model implemented with the Danfoss Kernel.

5.2 Code Generation & Compilation Conditions

The IAR visualSTATE® Coder options were set up to use function-pointer table for dispatching guards and actions. Data was initialized with C initializers (no system reinitialization allowed).

The SCOPE options used were: scope --release -cCF -cCstubs -cCdrv (essentially meaning flattening code generator, no debug information).

The H8 code was compiled with h8300-hms-gcc v. 3.3.2, options: h8300-hms-gcc -Os -static -DNDEBUG -fomit-frame-pointer -foptimize-sibling-calls -Xlinker --relax. The code was linked with newlib as the C library. A coff-h8300-hms executable was produced.

The ATMega16 code was compiled with avr-gcc v. 3.3.2, using options: -mmcu=atmega16 -mtiny-stack -Os -static -Wall -DNDEBUG -fomit-frame-pointer -foptimize-sibling-calls. The code was linked with lavr-libc as the C library. An elf-avr executable was produced.

The Danfoss Kernel for ATMega has been compiled using a makefile supplied with the kernel, optimizations set to -Os (optimize for size).

The sizes are reported for stripped binaries.

5.3 Manually implemented code

IAR visualSTATE® does not generate the main loop for its kernel. This gives increased flexibility, at the cost of transferring this effort to the developer. We present an example of simple main loop, which was used in the measurements in this section:

```
int main( void ) {
 SEM_ACTION_EXPRESSION_TYPE ActionExpressNo;
 SEM_EVENT_TYPE EventNo = SE_RESET;
 SEM_Init();             /* initialize kernel */
 SEM_InitSignalQueue(); /* initialize signal queue */

while(1) {
  /* Fire event */
  if ( SEM_Deduct( EventNo ) != SES_OKAY ) break;
  /* Compute System Reaction */
  while (SEM_GetOutput(&ActionExpressNo) == SES_FOUND)
     SEM_Action( ActionExpressNo );
  /* Advance system's state */
  if (SEM_NextState() != SES_OKAY) break;
  /* Sense next event from the environment */
  EventNo = (SEM_EVENT_TYPE)Sample;
} }
```

5.4 Project Structure

Proposed project structure:

```
|--api /* static IAR visualSTATE® libraries */
|  |-- SEMLibB.c /* API implementation (''kernel'') */
|  |-- SEMLibB.h /* IAR visualSTATE® API prototypes */
|  '-- VSTypes.h /* definitons of IAR visualSTATE® types */
|
|--code /* files generated with IAR visualSTATE® coder */
|  |-- EKCThermostat.c       /* transition tables */
|  |-- EKCThermostat.h
|  |-- EKCThermostatAction.h/* action functions types */
|  |-- EKCThermostatData.c /* model code&internal data */
|  |-- EKCThermostatData.h
|  |-- SEMBDef.h
|  '-- SEMTypes.h
|
|--driver.c /* hand-made: main loop,actions, drivers,... */
'--Makefile
```

The `api` contains the kernel, the `code` directory contains the generated code. Main directory contains user-written code.

5.5 Experiences

The conclusion is that the code generated by IAR visualSTATE® is bigger than expected and should be improved. Four areas of potential improvement have been identified:

- Thorough examination of the impact on efficiency by refactoring model by applying more event-driven approach, eliminating signals and exchanging guards on variables with events. This should be give a high priority in the next phase.

- Improvement in the kernel implementation, by porting the SCOPE kernel directly to ATMEL platform, or to the Danfoss kernel.
- Improvement in the code generator, by applying more research based optimizations to the existing tools (this effort is already on its way).
- Improvement in the integration with the EKC project (for instance sharing of parameters memory space, so it can be directly used from the model code).

All these four areas are potential fields of interest in the project.

6 Code Execution

The code generated from IAR visualSTATE® is ANSI compliant. It can be compiled on a workstation and executed, provided an adequate main loop and drivers are made available.

We have implemented such a simple setup for testing the EKC model on a PC, using a command-driven interface. Such execution may also be referred to as simulation, although more precisely this the execution of the actual model code, only compiled for a different platform. The interface was very simple (about 250 lines C program), nevertheless similar interfaces may be conveniently used in executing automatic test sequences in early phases, using a workstation (may be faster and cheaper than testing directly on the platform).

7 Test Generation

The model can also be used for automatically generating test cases for certain desired observations to be made on the system under test. Most model checking tools have a facility for producing traces or witnesses that explain the necessary steps to be taken to bring the model to a desired state. It is even possible to configure most tools to produce the shortest such trace.

In many cases this trace can be interpreted as a test case. Using this facility it is also possible to generate a set of test sequences that traverses all transition of the model thereby obtaining the model based testing dual to code based testing with statement coverage.

IAR visualSTATE® does not at present have such a trace generation facility, but one is being developed in collaboration with CISS. To demonstrate the usefulness of such a feature we have developed an equivalent model using a different notation and tool, namely Uppaal, and used the trace generation facility of Uppaal to synthesize a test case. We have annotated this model with an extra array of bits containing a bit for each interesting transition in the model, such that each bit is set when the corresponding transition is executed.

We give one short example of a test produced using this method: The goal of the example test is to switch cooling of as done by transition number 0 going from state *CoolerOn* to state *CoolerOff*, i.e., we search for a trace where $\texttt{bit}[0] == 1$. A condensed version of the *shortest trace* produced by Uppaal where irrelevant details have been manually removed is listed below. In this particular instance of the EKC model, *ThCutOut*=-1, *S4Percent*=100.

```
Delay: 1 //1

State:
GlobalTime=1
input #0:  DefrostActive=0 S3Error=0 S3Temp=0 S4Error=0 S4Temp=0
           StandbyActive=0
output #0: CoolingOn=0 HighAlarm=0 LowAlarm=0 ThTemp=0

Transitions: //2,3,4
  Sampler._id30->Sampler._id29 { 1, tau, S3Error := !S3Error }

State:
GlobalTime=1
input #1:  DefrostActive=0 S3Error=1 S3Temp=0 S4Error=0 S4Temp=0
           StandbyActive=0
output #1: CoolingOn=1 HighAlarm=0 LowAlarm=0 ThTemp=0

Transitions: //5 + // 6+7+8
  Delay: 1
  Sampler._id30->Sampler._id29 { 1, tau, S4Temp := CoolingOn &&
                  S4Temp > -maxtemp ? S4Temp - 1 : (S4Temp + 1) % maxtemp }


State:
GlobalTime=2
input #2:  DefrostActive=0 S3Error=1 S3Temp=0 S4Error=0 S4Temp=-1
           StandbyActive=0
output #2: CoolingOn=1 HighAlarm=0 LowAlarm=0 ThTemp=-1

Transitions: //9 + 10+11+12
  Delay: 1
  Sampler._id30->Sampler._id29 { 1, tau, S4Temp := CoolingOn &&
                 S4Temp > -maxtemp ? S4Temp - 1 : (S4Temp + 1) % maxtemp }


State:
GlobalTime=3
bits[0]=1 bits[1]=0 bits[2]=0 bits[3]=0 bits[4]=0 bits[5]=0 bits[6]=0
bits[7]=0 bits[8]=0 bits[9]=0 bits[10]=0 bits[11]=0 bits[12]=0
bits[13]=0 bits[14]=0 bits[15]=0 bits[16]=0 bits[17]=1
DutyTimer.limit=0 HighTimer.limit=0 LowTimer.limit=0 #depth=12

input #3:  DefrostActive=0 S3Error=1 S3Temp=0 S4Error=0 S4Temp=-2
           StandbyActive=0
output #3: CoolingOn=0 HighAlarm=0 LowAlarm=0 ThTemp=-2
```

The test recipe produced by Uppaal can be read as:

1. The preconditions for the test is defined by input/output vector #0
2. wait 1 time unit and set S3Error
   (a) give input vector #1: *S3Error==1*
   (b) expect output vector #1: *CoolingOn==1*
3. wait 1 time unit and decrease S4Temp
   (a) give input vector #2: *S4Temp==-1*
   (b) expect output vector #2: *ThTemp==-1*
4. waits 1 time unit and decrease S4Temp
   (a) give input vector #3: *S4Temp==-2*
   (b) expect output vector #3: *CoolingOn==0* and *ThTemp==-2*

It can be noted that in order to visit transition 0 also transition 17 will be taken, i.e. the number of test cases needed to cover all transitions will normally be much smaller than the number of transitions in the model.

8 Conclusions

Our experiences shows that the IAR visualSTATE® modeling language is very expressive, and could be used to create an elegant, reasonable compact model that accurately reflects the behavior intended by the designers, but that it has a number of tool specific limitations.

The IAR visualSTATE® verifier was useful, but an internal error was found that prevented all checks to be performed. It is our impression that the IAR visualSTATE® validator is a useful tool for functional testing at early design stages, and some errors in the model was detected this way.

The code generated by IAR visualSTATE® appear rather big for the given model. A thorough investigation as to the impact of the choice between ”shared-variable” versus ”event-driven” modeling on code size should be made.

The main feedback from DCD engineers was that the model was understandable and reflects a realistic part of an industrial cooler. However, they do not like the complex guards caused by the shared variable environment integration, and would prefer a pure event driven model. The code generated by IAR visualSTATE® is too large for their target platform, and must be improved if automatic code-generation is to be applied.

Overall DCD found that the IAR visualSTATE® approach appeared promising as a platform for model based development using formalized modeling and documentation, simulation and testing, and coding, and if sufficient improvements can be made, also for automatic code generation. Future possibilities of automatic of semi-automatic test generation from the model would be particulary helpfull to DCD.

A Danfoss EKC Requirements Specification

Visual State trial project

EKC thermostat Specification

Abstract

Specification of a trial project for Visual State modelling, integration and simulation

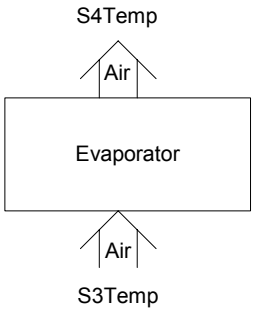
Table of Contents

1. Thermostat specification

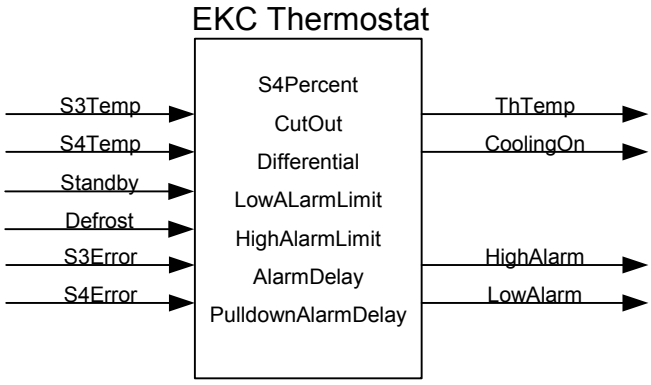
1.1 Description

The thermostat function in an EKC controller determines whether cooling is to take place or not. The thermostat uses a calculated temperature value, a cut out value and a differential to determine whether the thermostat is to cut in or out.

The physical system



The thermostat



The thermostat is overruled to cut out if the controller is put into standby mode/state or if defrost mode/state is entered.

If the thermostat temperature is above an upper limit, a high temperature alarm state is raised after a time delay. If the high alarm state is entered before the first cut out after 1) power on 2) leaving stand-by mode/state 3) leaving defrost mode/state another time delay, pull-down time delay, apply.

If the thermostat temperature is below a lower limit, a low temperature alarm state is raised after a time delay.

In case of sensor error (short or open circuit), the thermostat cut-in/out is not determined from the used temperature input, but is calculated from the cut in/out history (emergency cooling):

1) If the count of cut outs is below 50, the cut-in/out sequence is calculated from a 30% on/70% off duty cycle with a period of 20 minutes.

2) Otherwise the cut-in/out duty cycle is calculated from minutes in cut-in and minutes in cut out (excluding standby and defrost cut-out time) again with a period of 20 minutes.

1.2 Input

Float flS3Temp	Range –200.0, 200.0, temperature from sensor placed in the air stream before the evaporator
Float flS4Temp	Range –200.0, 200.0, temperature from sensor placed in the air stream after the evaporator
Boolean bStandbyActive	True if standby mode/state active
Boolean bDefrostActive	True if in defrost sequence
Boolean bS3Error	The value of S3Temp is not valid due to a sensor error detected
Boolean bS3Error	The value of S4Temp is not valid due to a sensor error detected

1.3 Output

Float flThTemp	Range –200.0, 200.0, calculated thermostat temperature. Used to determine cut-in/out and alarm state. FlThTemp = (100-iS4Percent)*flS3Temp +iS4Percent*flS4Temp In case of sensor error S3/S4 flThTemp=180.0, but: If iS4Percent=100 and bS3Error then flThTemp=flS4Temp If iS4Percent=0 and bS4Error then flThTemp=flS3Temp
Boolean bCoolingOn	True if cooling is to be on. False if cooling is to be off, standby mode/state or defrost mode/state
Boolean bHighAlarm	True if high limit alarm conditions are met
Boolean bLowAlarm	True if low limit alarm conditions are met

1.4 Settings

Int iS4Percent	Range 0, 100, weighting between S4/S3 sensor
Float flThCutOut	Range –200.0, 200.0, the thermostat cut out limit
Float flThDifferential	Range 0.0, 50.0, the differential value (hysteresis) defining the cut in value as cut out value plus hysteresis.
Float flHighAlarmLimit	Range –200.0, 200.0, When the thermostat value goes above the limit, an

	high limit alarm must be set after a time delay
Float flLowAlarmLimit	Range –200.0, 200.0, When the thermostat value goes below the limit, an low limit alarm must be set after a time delay
Int iAlarmDelay	Range 0, 30, the alarm delay in minutes before a high/low alarm is set
Int iPulldownAlarmDelay	Range 0, 60, the alarm delay in minutes before a high/low alarm is set. The delay replaces iAlarmDelay before the first thermostat cut out after power on, leaving stand-by mode or leaving defrost mode.

1.5 Parameter interface

Parameter are organized in an array structure, one part placed in ROM and one part placed in RAM. The structure defines all options and properties regarding a parameter. Parameter types are of 8, 16, 32 bit size e.g. enum, char, int, long and float.

An example of the parameter interface used today is to be found here:


DATABASE.C


DATABASE.H

The application software today uses constant pointers declared to point to the RAM part:
*pdbS3Temp = 10;
*pddS4Temp = 15;

B IAR visualSTATE® generated documentation

Project

Table of contents

Model design

IAR visualSTATE Signature Generator: "50"

Project Signature: "7ee2 7035 ea2a ea1a ce81 8caf"

Chart

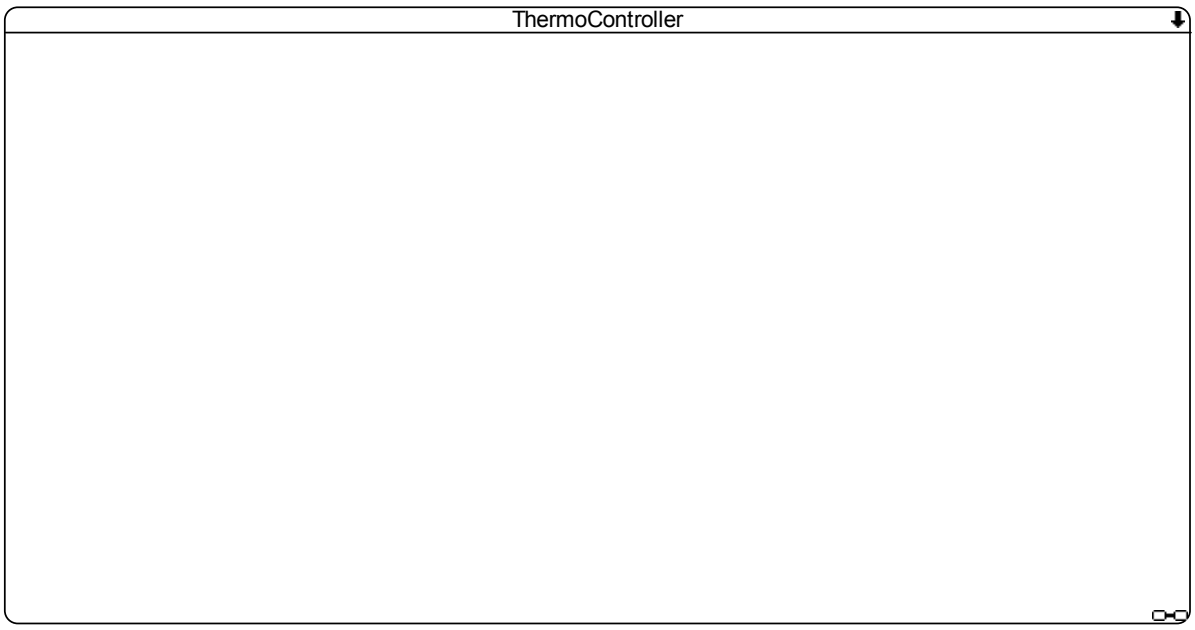
Hierarchy

- Project
  - Elements
  - EKCThermostat
    - ThermoController

Elements

EKCThermostat

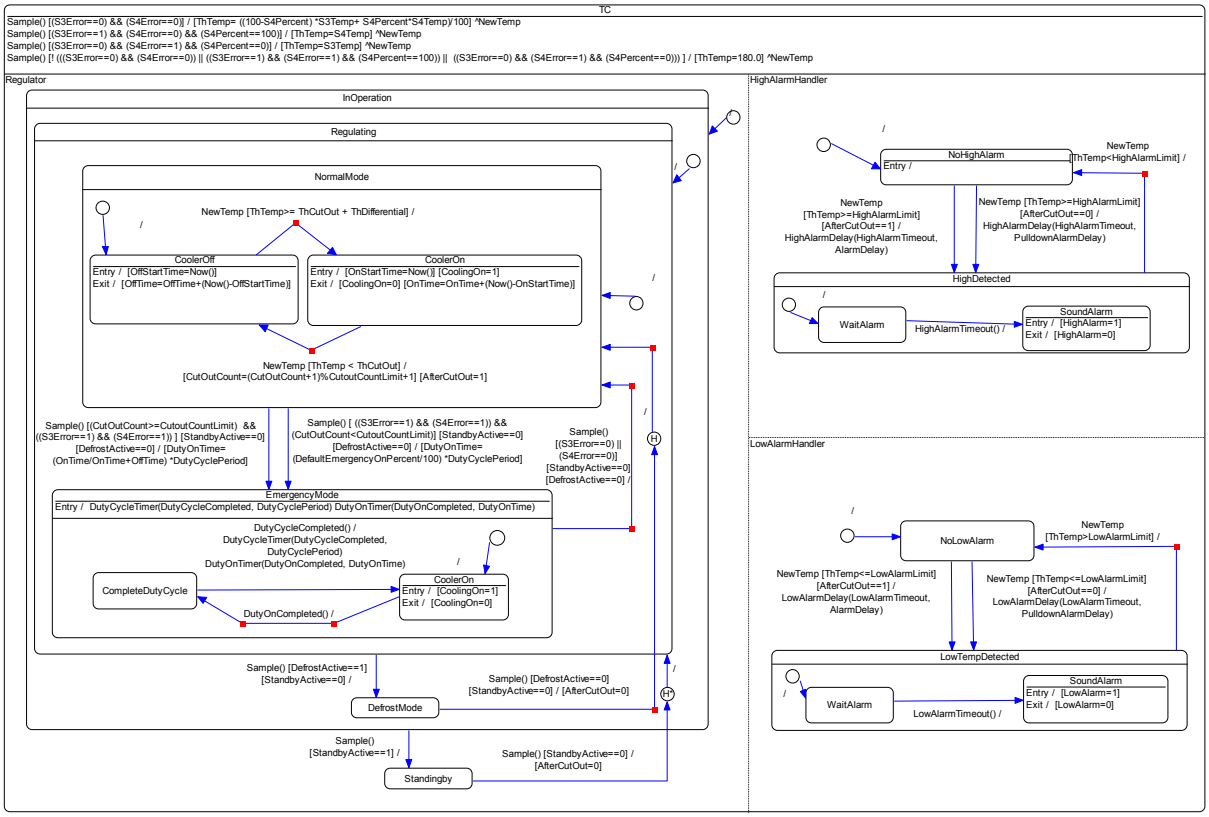
Chart



ThermoController (top)

Path: EKCThermostat

Chart



Hierarchy

- ThermoController
  - Elements
  - TC
    - Regulator
      - Standingby
      - InOperation
        - Regulating
          - NormalMode
            - CoolerOff
            - CoolerOn
          - EmergencyMode
            - CompleteDutyCycle
            - CoolerOn
        - DefrostMode
    - HighAlarmHandler
      - NoHighAlarm
      - HighDetected
        - WaitAlarm
        - SoundAlarm
    - LowAlarmHandler

- NoLowAlarm
- LowTempDetected
  - WaitAlarm
  - SoundAlarm

Elements

Events

- DutyCycleCompleted ()
- DutyOnCompleted ()
- HighAlarmTimeout ()
- LowAlarmTimeout ()
- Sample ()

Action functions

- extern VS_INT Now ()
- extern VS_VOID DutyCycleTimer (VS_UINT event, VS_UINT ticks)
- extern VS_VOID DutyOnTimer (VS_UINT event, VS_UINT ticks)
- extern VS_VOID HighAlarmDelay (VS_UINT event, VS_UINT ticks)
- extern VS_VOID LowAlarmDelay (VS_UINT event, VS_UINT ticks)

External variables

- VS_UINT8 AlarmDelay = 50
- VS_BOOL CoolingOn = 0
- VS_BOOL DefrostActive = 0
- VS_BOOL HighAlarm = 0
- VS_INT HighAlarmLimit = 15
- VS_BOOL LowAlarm = 0
- VS_INT LowAlarmLimit = 0
- VS_UINT8 PulldownAlarmDelay = 200
- VS_INT S3Error = 0
- VS_INT S3Temp = 0
- VS_INT S4Error = 0
- VS_INT S4Percent = 50
- VS_INT S4Temp = 0
- VS_BOOL StandbyActive = 0
- VS_INT ThCutOut = 5
- VS_INT ThDifferential = 2
- VS_INT ThTemp = 0

Internal variables

- VS_BOOL AfterCutOut = 0
- VS_INT CutOutCount = 0
- VS_INT DutyOnTime = 0
- VS_INT OffStartTime = 0
- VS_INT OffTime = 0
- VS_INT OnStartTime = 0
- VS_INT OnTime = 0

Signals

- NewTemp

Constants

- VS_UINT8 CutoutCountLimit = 50
- VS_INT DefaultEmergencyOnPercent = 30
- VS_INT DutyCyclePeriod = 20

Transitions

- Standingby -> Sample() [StandbyActive == 0] / [AfterCutOut = 0] -> _DeepHistoryState0;
- InOperation -> Sample() [StandbyActive == 1] / -> Standingby;
- NoHighAlarm -> NewTemp [ThTemp >= HighAlarmLimit] [AfterCutOut == 1] / [HighAlarmDelay (HighAlarmTimeout, AlarmDelay)] -> HighDetected;
- NoHighAlarm -> NewTemp [ThTemp >= HighAlarmLimit] [AfterCutOut == 0] / [HighAlarmDelay (HighAlarmTimeout, PulldownAlarmDelay)] -> HighDetected;
- HighDetected -> NewTemp [ThTemp < HighAlarmLimit] / -> NoHighAlarm;
- NoLowAlarm -> NewTemp [ThTemp <= LowAlarmLimit] [AfterCutOut == 1] / [LowAlarmDelay (LowAlarmTimeout, AlarmDelay)] -> LowTempDetected;
- NoLowAlarm -> NewTemp [ThTemp <= LowAlarmLimit] [AfterCutOut == 0] / [LowAlarmDelay (LowAlarmTimeout, PulldownAlarmDelay)] -> LowTempDetected;
- LowTempDetected -> NewTemp [ThTemp > LowAlarmLimit] / -> NoLowAlarm;
- Regulating -> Sample() [DefrostActive == 1] [StandbyActive == 0] / -> DefrostMode;
- DefrostMode -> Sample() [DefrostActive == 0] [StandbyActive == 0] / [AfterCutOut = 0] -> _ShallowHistoryState0;
- NormalMode -> Sample() [(CutOutCount >= CutoutCountLimit) && ((S3Error == 1) && (S4Error == 1))] [StandbyActive == 0] [DefrostActive == 0] / [DutyOnTime = (OnTime / OnTime + OffTime) * DutyCyclePeriod] -> EmergencyMode;
- NormalMode -> Sample() [((S3Error == 1) && (S4Error == 1)) && (CutOutCount < CutoutCountLimit)] [StandbyActive == 0] [DefrostActive == 0] / [DutyOnTime = (DefaultEmergencyOnPercent / 100) * DutyCyclePeriod] -> EmergencyMode;
- EmergencyMode -> Sample() [(S3Error == 0) || (S4Error == 0)] [StandbyActive == 0] [DefrostActive == 0] / -> NormalMode;
- CoolerOff -> NewTemp [ThTemp >= ThCutOut + ThDifferential] / -> CoolerOn;
- CoolerOn -> NewTemp [ThTemp < ThCutOut] / [CutOutCount = (CutOutCount + 1) % CutoutCountLimit + 1] [AfterCutOut = 1] -> CoolerOff;
- CompleteDutyCycle -> DutyCycleCompleted() / [DutyCycleTimer (DutyCycleCompleted, DutyCyclePeriod)] [DutyOnTimer (DutyOnCompleted, DutyOnTime)] -> CoolerOn;
- CoolerOn -> DutyOnCompleted() / -> CompleteDutyCycle;
- WaitAlarm -> HighAlarmTimeout() / -> SoundAlarm;
- WaitAlarm -> LowAlarmTimeout() / -> SoundAlarm;

States

TC

Path: EKCThermostat:ThermoController

Reactions and default transitions

- Sample() [(S3Error == 0) && (S4Error == 0)] / [ThTemp = ((100 - S4Percent) * S3Temp + S4Percent * S4Temp) / 100] ^NewTemp;
- Sample() [(S3Error == 1) && (S4Error == 0) && (S4Percent == 100)] / [ThTemp = S4Temp] ^NewTemp;
- Sample() [(S3Error == 0) && (S4Error == 1) && (S4Percent == 0)] / [ThTemp = S3Temp] ^NewTemp;
- Sample() [!(((S3Error == 0) && (S4Error == 0)) || ((S3Error == 1) && (S4Error == 1) && (S4Percent == 100)) || ((S3Error == 0) && (S4Error == 1) && (S4Percent == 0)))] / [ThTemp = 180.0] ^NewTemp;

Regulator

Path: EKCThermostat:ThermoController.TC

Standingby

Path: EKCThermostat:ThermoController.TC.Regulator

InOperation

Path: EKCThermostat:ThermoController.TC.Regulator

Regulating

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation

NormalMode

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating

CoolerOff

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating.NormalMode

Reactions and default transitions

- Entry / [OffStartTime = Now ()];
- Exit / [OffTime = OffTime + (Now () - OffStartTime)];

CoolerOn

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating.NormalMode

Reactions and default transitions

- Entry / [OnStartTime = Now ()] [CoolingOn = 1];
- Exit / [CoolingOn = 0] [OnTime = OnTime + (Now () - OnStartTime)];
- _InitialState0 -> / -> CoolerOff;

EmergencyMode

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating

Reactions and default transitions

- Entry / [DutyCycleTimer (DutyCycleCompleted, DutyCyclePeriod)] [DutyOnTimer (DutyOnCompleted, DutyOnTime)];

CompleteDutyCycle

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating.EmergencyMode

CoolerOn

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation.Regulating.EmergencyMode

Reactions and default transitions

- Entry / [CoolingOn = 1];
- Exit / [CoolingOn = 0];
- _InitialState1 -> / -> CoolerOn;
- _ShallowHistoryState0 -> / -> NormalMode;
- _InitialState2 -> / -> NormalMode;

DefrostMode

Path: EKCThermostat:ThermoController.TC.Regulator.InOperation

Reactions and default transitions

- _DeepHistoryState0 -> / -> Regulating;
- _InitialState3 -> / -> Regulating;
- _InitialState4 -> / -> InOperation;

HighAlarmHandler

Path: EKCThermostat:ThermoController.TC

NoHighAlarm

Path: EKCThermostat:ThermoController.TC.HighAlarmHandler

HighDetected

Path: EKCThermostat:ThermoController.TC.HighAlarmHandler

WaitAlarm

Path: EKCThermostat:ThermoController.TC.HighAlarmHandler.HighDetected

SoundAlarm

Path: EKCThermostat:ThermoController.TC.HighAlarmHandler.HighDetected

Reactions and default transitions

- Entry / [HighAlarm = 1];
- Exit / [HighAlarm = 0];
- _InitialState5 -> / -> WaitAlarm;
- _InitialState6 -> / -> NoHighAlarm;

LowAlarmHandler

Path: EKCThermostat:ThermoController.TC

NoLowAlarm

Path: EKCThermostat:ThermoController.TC.LowAlarmHandler

LowTempDetected

Path: EKCThermostat:ThermoController.TC.LowAlarmHandler

WaitAlarm

Path: EKCThermostat:ThermoController.TC.LowAlarmHandler.LowTempDetected

SoundAlarm

Path: EKCThermostat:ThermoController.TC.LowAlarmHandler.LowTempDetected

Reactions and default transitions

- Entry / [LowAlarm = 1];
- Exit / [LowAlarm = 0];
- _InitialState7 -> / -> WaitAlarm;
- _InitialState8 -> / -> NoLowAlarm;

Model test

Model interface

Events

Event	Path
DutyCycleCompleted	EKCThermostat:ThermoController
DutyOnCompleted	EKCThermostat:ThermoController
HighAlarmTimeout	EKCThermostat:ThermoController
LowAlarmTimeout	EKCThermostat:ThermoController
Sample	EKCThermostat:ThermoController

Action functions

Action function	Path
DutyCycleTimer	EKCThermostat:ThermoController
DutyOnTimer	EKCThermostat:ThermoController
HighAlarmDelay	EKCThermostat:ThermoController
LowAlarmDelay	EKCThermostat:ThermoController
Now	EKCThermostat:ThermoController

External variables

External variable	Path
AlarmDelay	EKCThermostat:ThermoController
CoolingOn	EKCThermostat:ThermoController
DefrostActive	EKCThermostat:ThermoController
HighAlarm	EKCThermostat:ThermoController
HighAlarmLimit	EKCThermostat:ThermoController
LowAlarm	EKCThermostat:ThermoController
LowAlarmLimit	EKCThermostat:ThermoController
PulldownAlarmDelay	EKCThermostat:ThermoController
S3Error	EKCThermostat:ThermoController
S3Temp	EKCThermostat:ThermoController
S4Error	EKCThermostat:ThermoController
S4Percent	EKCThermostat:ThermoController
S4Temp	EKCThermostat:ThermoController
StandbyActive	EKCThermostat:ThermoController

ThCutOut	EKCThermostat:ThermoController
ThDifferential	EKCThermostat:ThermoController
ThTemp	EKCThermostat:ThermoController

Constants

Constant	Path
CutoutCountLimit	EKCThermostat:ThermoController
DefaultEmergencyOnPercent	EKCThermostat:ThermoController
DutyCyclePeriod	EKCThermostat:ThermoController

Implementation

Pseudo code

```
Project Project
{
  SignalQueueOverflowBehavior: errorIfFull;
  Elements
  {
  }
  System EKCThermostat
  {
    Instances: 1;
    SignalQueueLength: 5;
    TopState ThermoController;
  }
}

TopState EKCThermostat:ThermoController
{
  Elements
  {
    Events
    {
      DutyCycleCompleted ();
      DutyOnCompleted ();
      HighAlarmTimeout ();
      LowAlarmTimeout ();
      Sample ();  /* A new set of environment variables are available, and the
controller
needs to change state */
    }
    ActionFunctions
    {
      extern VS_INT Now ();  /* Returns current global real-time (tics) since
system start */
      extern VS_VOID DutyCycleTimer (VS_UINT event, VS_UINT ticks);
      extern VS_VOID DutyOnTimer (VS_UINT event, VS_UINT ticks);
      extern VS_VOID HighAlarmDelay (VS_UINT event, VS_UINT ticks);
      extern VS_VOID LowAlarmDelay (VS_UINT event, VS_UINT ticks);
    }
    ExternalVariables
    {
      VS_UINT8 AlarmDelay = 50;  /* INPUT SETTING
Range 0, 30
the alarm delay in minutes before a high/low alarm is set */
      VS_BOOL CoolingOn = 0;  /* OUTPUT
Danfoss:
True if cooling is to be on.
False if cooling is to be off, standby mode/state or defrost mode/state */
      VS_BOOL DefrostActive = 0;  /* INPUT: True when defrost mode is requested

Danfoss: True if in defrost sequence */
      VS_BOOL HighAlarm = 0;  /* OUTPUT:

Danfoss:
True if high limit alarm conditions are met */
      VS_INT HighAlarmLimit = 15;  /* INPUT SETTING
Danfoss
Range -200.0, 200.0,
When the thermostat value goes above the limit, an high limit alarm must be set
after a time delay */
      VS_BOOL LowAlarm = 0;  /* OUTPUT
Danfoss:

True if low limit alarm conditions are met */
      VS_INT LowAlarmLimit = 0;  /* INPUT SETTING
Danfoss:
```

```
Range -200.0, 200.0,
When the thermostat value goes below the limit, an low limit alarm must be set
after a time delay */
      VS_UINT8 PulldownAlarmDelay = 200;  /* INPUT SETTING

Range 0, 60,
the alarm delay in minutes before a high/low alarm is set.
The delay replaces iAlarmDelay before the first thermostat cut out after power on,
leaving stand-by mode or leaving defrost mode. */
      VS_INT S3Error = 0;  /* INPUT
True when an error has been detected on Sensor 3 (incomming air temperature)
Danfoss:
The value of S3Temp is not valid due to a sensor error detected */
      VS_INT S3Temp = 0;  /* INPUT: Incomming Air Temperature

Danfoss:
Range -200.0, 200.0, T
emperature from sensor placed in the air stream before the evaporator */
      VS_INT S4Error = 0;  /* INPUT
True when an error has been detected on Sensor 3 (incomming air temperature)
Danfoss:
The value of S4Temp is not valid due to a sensor error detected */
      VS_INT S4Percent = 50;  /* INPUT SETTING
Danfoss:
Range 0, 100,
weighting between S4/S3 sensor */
      VS_INT S4Temp = 0;  /* INPUT: Outcomimng air temperature

Danfoss:
Range -200.0, 200.0,
temperature from sensor placed in the air stream after the evaporator */
      VS_BOOL StandbyActive = 0;  /* INPUT True when standby is requested
Danfoss:
True if standby mode/state active */
      VS_INT ThCutOut = 5;  /* INPUT SETTING
Defines threshold for disabling cooling

Danfoss:
Range -200.0, 200.0,
the thermostat cut out limit */
      VS_INT ThDifferential = 2;  /* INPUT SETTING

Danfoss:
Range 0.0, 50.0,
the differential value (hysteresis) defining the cut in value as cut out value plus
hysteresis. */
      VS_INT ThTemp = 0;  /* OUTPUT/INTERNAL
Calculated Thermostat temperature

Danfoss:
Range -200.0, 200.0,
calculated thermostat temperature. Used to determine cut-in/out and alarm state.
FlThTemp = (100-iS4Percent)*flS3Temp +iS4Percent*flS4Temp
In case of sensor error S3/S4 flThTemp=180.0, but:
If iS4Percent=100 and bS3Error then flThTemp=flS4Temp
If iS4Percent=0 and bS4Error then flThTemp=flS3Temp */
    }
    InternalVariables
    {
      VS_BOOL AfterCutOut = 0;
      VS_INT CutOutCount = 0;
      VS_INT DutyOnTime = 0;
      VS_INT OffStartTime = 0;  /* Time at which a state was entered */
      VS_INT OffTime = 0;
      VS_INT OnStartTime = 0;
      VS_INT OnTime = 0;
    }
    Signals
```

```
    {
      NewTemp;  /* A new Calculated Temperature Has been calculated */
    }
    Constants
    {
      VS_UINT8 CutoutCountLimit = 50;  /* Number of cutouts determines wheter to
use historically computed emergency cooling duty cycle or
default 30/70 duty cycle.

In case of sensor error (short or open circuit), the thermostat cut-in/out is not
determined from the used temperature input, but is calculated from the cut in/out
history (emergency cooling):
1)     If the count of cut outs is below 50, the cut-in/out sequence is calculated
from a 30% on/70% off duty cycle with a period of 20 minutes.
2)     Otherwise the cut-in/out duty cycle is calculated from minutes in cut-in and
minutes in cut out (excluding standby and defrost cut-out time) again with a period
of 20 minutes.
 */
      VS_INT DefaultEmergencyOnPercent = 30;  /* The fraction of a duty cycle
period where the cooling is required to be on.   */
      VS_INT DutyCyclePeriod = 20;  /* Length of a duty cycle in emergency mode
 */
    }
  }
  Transitions
  {
    ThermoController.TC.Standingby -> Sample() [StandbyActive==0] / [AfterCutOut=0]
-> ThermoController.TC.InOperation._DeepHistoryState0;
    ThermoController.TC.InOperation -> Sample() [StandbyActive==1] / ->
ThermoController.TC.Standingby;
    ThermoController.TC.NoHighAlarm -> NewTemp [ThTemp>=HighAlarmLimit]
[AfterCutOut==1] / [HighAlarmDelay (HighAlarmTimeout, AlarmDelay)] ->
ThermoController.TC.HighDetected;
    ThermoController.TC.NoHighAlarm -> NewTemp [ThTemp>=HighAlarmLimit]
[AfterCutOut==0] / [HighAlarmDelay (HighAlarmTimeout, PulldownAlarmDelay)] ->
ThermoController.TC.HighDetected;
    ThermoController.TC.HighDetected -> NewTemp [ThTemp<HighAlarmLimit] / ->
ThermoController.TC.NoHighAlarm;
    ThermoController.TC.NoLowAlarm -> NewTemp [ThTemp<=LowAlarmLimit]
[AfterCutOut==1] / [LowAlarmDelay (LowAlarmTimeout, AlarmDelay)] ->
ThermoController.TC.LowTempDetected;
    ThermoController.TC.NoLowAlarm -> NewTemp [ThTemp<=LowAlarmLimit]
[AfterCutOut==0] / [LowAlarmDelay (LowAlarmTimeout, PulldownAlarmDelay)] ->
ThermoController.TC.LowTempDetected;
    ThermoController.TC.LowTempDetected -> NewTemp [ThTemp>LowAlarmLimit] / ->
ThermoController.TC.NoLowAlarm;
    ThermoController.TC.InOperation.Regulating -> Sample() [DefrostActive==1]
[StandbyActive==0] / -> ThermoController.TC.InOperation.DefrostMode;
    ThermoController.TC.InOperation.DefrostMode -> Sample() [DefrostActive==0]
[StandbyActive==0] / [AfterCutOut=0] ->
ThermoController.TC.InOperation.Regulating._ShallowHistoryState0;
    ThermoController.TC.InOperation.Regulating.NormalMode -> Sample()
[(CutOutCount>=CutoutCountLimit)  && ((S3Error==1) && (S4Error==1)) ]
[StandbyActive==0] [DefrostActive==0] / [DutyOnTime= (OnTime/OnTime+OffTime)
*DutyCyclePeriod] -> ThermoController.TC.InOperation.Regulating.EmergencyMode;
    ThermoController.TC.InOperation.Regulating.NormalMode -> Sample() [
((S3Error==1) && (S4Error==1)) && (CutOutCount<CutoutCountLimit)]
[StandbyActive==0] [DefrostActive==0] / [DutyOnTime=
(DefaultEmergencyOnPercent/100) *DutyCyclePeriod] ->
ThermoController.TC.InOperation.Regulating.EmergencyMode;
    ThermoController.TC.InOperation.Regulating.EmergencyMode -> Sample()
[(S3Error==0) || (S4Error==0)] [StandbyActive==0] [DefrostActive==0] / ->
ThermoController.TC.InOperation.Regulating.NormalMode;
    ThermoController.TC.InOperation.Regulating.NormalMode.CoolerOff -> NewTemp
[ThTemp>= ThCutOut + ThDifferential] / ->
ThermoController.TC.InOperation.Regulating.NormalMode.CoolerOn;
    ThermoController.TC.InOperation.Regulating.NormalMode.CoolerOn -> NewTemp
[ThTemp < ThCutOut] / [CutOutCount=(CutOutCount+1)%CutoutCountLimit+1]
```

```
[AfterCutOut=1] -> ThermoController.TC.InOperation.Regulating.NormalMode.CoolerOff;
    ThermoController.TC.InOperation.Regulating.EmergencyMode.CompleteDutyCycle ->
DutyCycleCompleted() / [DutyCycleTimer (DutyCycleCompleted, DutyCyclePeriod)]
[DutyOnTimer (DutyOnCompleted, DutyOnTime)] ->
ThermoController.TC.InOperation.Regulating.EmergencyMode.CoolerOn;
    ThermoController.TC.InOperation.Regulating.EmergencyMode.CoolerOn ->
DutyOnCompleted() / ->
ThermoController.TC.InOperation.Regulating.EmergencyMode.CompleteDutyCycle;
    ThermoController.TC.HighDetected.WaitAlarm -> HighAlarmTimeout() / ->
ThermoController.TC.HighDetected.SoundAlarm;
    ThermoController.TC.LowTempDetected.WaitAlarm -> LowAlarmTimeout() / ->
ThermoController.TC.LowTempDetected.SoundAlarm;
  }
  CompositeState TC
  {
    Sample() [(S3Error==0) && (S4Error==0)] / [ThTemp= ((100-S4Percent) *S3Temp+
S4Percent*S4Temp)/100] ^NewTemp;
    Sample() [(S3Error==1) && (S4Error==0) && (S4Percent==100)] / [ThTemp=S4Temp]
^NewTemp;
    Sample() [(S3Error==0) && (S4Error==1) && (S4Percent==0)] / [ThTemp=S3Temp]
^NewTemp;
    Sample() [! (((S3Error==0) && (S4Error==0)) || ((S3Error==1) && (S4Error==1) &&
(S4Percent==100)) ||  ((S3Error==0) && (S4Error==1) && (S4Percent==0))) ] /
[ThTemp=180.0] ^NewTemp;

    Region Regulator
    {
      SimpleState Standingby
      {
      }
      CompositeState InOperation
      {
        CompositeState Regulating
        {
          CompositeState NormalMode
          {
            SimpleState CoolerOff
            {
              Entry / [OffStartTime=Now()];
              Exit / [OffTime=OffTime+(Now()-OffStartTime)];
            }

            SimpleState CoolerOn
            {
              Entry / [OnStartTime=Now()] [CoolingOn=1];
              Exit / [CoolingOn=0] [OnTime=OnTime+(Now()-OnStartTime)];
            }
            ThermoController.TC.InOperation.Regulating.NormalMode._InitialState0 ->
/ -> ThermoController.TC.InOperation.Regulating.NormalMode.CoolerOff;
          }

          CompositeState EmergencyMode
          {
            Entry / [DutyCycleTimer (DutyCycleCompleted, DutyCyclePeriod)]
[DutyOnTimer (DutyOnCompleted, DutyOnTime)];

            SimpleState CompleteDutyCycle
            {
            }
            SimpleState CoolerOn
            {
              Entry / [CoolingOn=1];
              Exit / [CoolingOn=0];
            }
            ThermoController.TC.InOperation.Regulating.EmergencyMode._InitialState1
-> / -> ThermoController.TC.InOperation.Regulating.EmergencyMode.CoolerOn;
          }
          ThermoController.TC.InOperation.Regulating._ShallowHistoryState0 -> / ->
```

```
ThermoController.TC.InOperation.Regulating.NormalMode;
          ThermoController.TC.InOperation.Regulating._InitialState2 -> / ->
ThermoController.TC.InOperation.Regulating.NormalMode;
        }

        SimpleState DefrostMode
        {
        }
        ThermoController.TC.InOperation._DeepHistoryState0 -> / ->
ThermoController.TC.InOperation.Regulating;
        ThermoController.TC.InOperation._InitialState3 -> / ->
ThermoController.TC.InOperation.Regulating;
      }
      ThermoController.TC._InitialState4 -> / -> ThermoController.TC.InOperation;
    }

    Region HighAlarmHandler
    {
      SimpleState NoHighAlarm
      {
      }
      CompositeState HighDetected
      {
        SimpleState WaitAlarm
        {
        }
        SimpleState SoundAlarm
        {
          Entry / [HighAlarm=1];
          Exit / [HighAlarm=0];
        }
        ThermoController.TC.HighDetected._InitialState5 -> / ->
ThermoController.TC.HighDetected.WaitAlarm;
      }
      ThermoController.TC._InitialState6 -> / -> ThermoController.TC.NoHighAlarm;
    }

    Region LowAlarmHandler
    {
      SimpleState NoLowAlarm
      {
      }
      CompositeState LowTempDetected
      {
        SimpleState WaitAlarm
        {
        }
        SimpleState SoundAlarm
        {
          Entry / [LowAlarm=1];
          Exit / [LowAlarm=0];
        }
        ThermoController.TC.LowTempDetected._InitialState7 -> / ->
ThermoController.TC.LowTempDetected.WaitAlarm;
      }
      ThermoController.TC._InitialState8 -> / -> ThermoController.TC.NoLowAlarm;
    }
  }
}
```

Element lists

Events

Event	Path
DutyCycleCompleted	EKCThermostat:ThermoController
DutyOnCompleted	EKCThermostat:ThermoController
HighAlarmTimeout	EKCThermostat:ThermoController
LowAlarmTimeout	EKCThermostat:ThermoController
Sample	EKCThermostat:ThermoController

Action functions

Action function	Path
DutyCycleTimer	EKCThermostat:ThermoController
DutyOnTimer	EKCThermostat:ThermoController
HighAlarmDelay	EKCThermostat:ThermoController
LowAlarmDelay	EKCThermostat:ThermoController
Now	EKCThermostat:ThermoController

External variables

External variable	Path
AlarmDelay	EKCThermostat:ThermoController
CoolingOn	EKCThermostat:ThermoController
DefrostActive	EKCThermostat:ThermoController
HighAlarm	EKCThermostat:ThermoController
HighAlarmLimit	EKCThermostat:ThermoController
LowAlarm	EKCThermostat:ThermoController
LowAlarmLimit	EKCThermostat:ThermoController
PulldownAlarmDelay	EKCThermostat:ThermoController
S3Error	EKCThermostat:ThermoController
S3Temp	EKCThermostat:ThermoController
S4Error	EKCThermostat:ThermoController
S4Percent	EKCThermostat:ThermoController
S4Temp	EKCThermostat:ThermoController
StandbyActive	EKCThermostat:ThermoController

ThCutOut	EKCThermostat:ThermoController
ThDifferential	EKCThermostat:ThermoController
ThTemp	EKCThermostat:ThermoController

Internal variables

Internal variable	Path
AfterCutOut	EKCThermostat:ThermoController
CutOutCount	EKCThermostat:ThermoController
DutyOnTime	EKCThermostat:ThermoController
OffStartTime	EKCThermostat:ThermoController
OffTime	EKCThermostat:ThermoController
OnStartTime	EKCThermostat:ThermoController
OnTime	EKCThermostat:ThermoController

Signals

Signal	Path
NewTemp	EKCThermostat:ThermoController

Constants

Constant	Path
CutoutCountLimit	EKCThermostat:ThermoController
DefaultEmergencyOnPercent	EKCThermostat:ThermoController
DutyCyclePeriod	EKCThermostat:ThermoController

Index

Index entries: ACDEHILNOPRSTW

A

C

D

    - Destination state [p10]
- NormalMode [p8] (State)
    - Source state [p7]
    - Source state [p7]
    - Destination state [p7]
    - Destination state [p9]
    - Destination state [p9]
- Now [p6] (Action function declaration)
    - Action expression [p8]
    - Action expression [p8]
    - Action expression [p8]
    - Action expression [p8]

O

- OffStartTime [p6] (Internal variable)
    - Action expression [p8]
    - Action expression [p8]
- OffTime [p6] (Internal variable)
    - Action expression [p7]
    - Action expression [p8]
    - Action expression [p8]
- OnStartTime [p6] (Internal variable)
    - Action expression [p8]
    - Action expression [p8]
- OnTime [p6] (Internal variable)
    - Action expression [p7]
    - Action expression [p7]
    - Action expression [p8]
    - Action expression [p8]

P

- PulldownAlarmDelay [p6] (External variable)
    - Action expression [p7]
    - Action expression [p7]

R

- Regulating [p8] (State)
    - Source state [p7]
    - Destination state [p9]
    - Destination state [p9]

S

- S3Error [p6] (External variable)
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
- S3Temp [p6] (External variable)
    - Action expression [p7]

    - Action expression [p7]
- S4Error [p6] (External variable)
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
- S4Percent [p6] (External variable)
    - Action expression [p7]
    - Action expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
- S4Temp [p6] (External variable)
    - Action expression [p7]
    - Action expression [p7]
- Sample [p6] (Event)
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
    - Trigger [p7]
- SoundAlarm [p9] (State)
    - Destination state [p7]
- SoundAlarm [p10] (State)
    - Destination state [p7]
- StandbyActive [p6] (External variable)
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
- Standingby [p8] (State)
    - Source state [p7]
    - Destination state [p7]

T

- TC [p7] (State) (unreferenced)
- ThCutOut [p6] (External variable)

    - Guard expression [p7]
    - Guard expression [p7]
- ThDifferential [p6] (External variable)
    - Guard expression [p7]
- ThermoController [p4] (State) (unreferenced)
- ThTemp [p6] (External variable)
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Guard expression [p7]
    - Action expression [p7]
    - Action expression [p7]
    - Action expression [p7]
    - Action expression [p7]

W

- WaitAlarm [p10] (State)
    - Source state [p7]
    - Destination state [p10]
- WaitAlarm [p9] (State)
    - Source state [p7]
    - Destination state [p9]

Recent BRICS Report Series Publications

RS-03-48 Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, and Andrzej Wasowski. *Danfoss EKC Trial Project Deliverables*. December 2003. 53 pp.

RS-03-47 Hans Hüttel and Jiří Srba. *Recursive Ping-Pong Protocols*. December 2003. To appear in the proceedings of 2004 IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues in the Theory of Security (WITS'04).

RS-03-46 Philipp Gerhardy. *The Role of Quantifier Alternations in Cut Elimination*. December 2003. 10 pp. Extends paper appearing in Baaz and Makowsky, editors, *European Association for Computer Science Logic: 17th International Workshop*, CSL '03 Proceedings, LNCS 2803, 2003, pages 212-225.

RS-03-45 Peter Bro Miltersen, Jaikumar Radhakrishnan, and Ingo Wegener. *On converting CNF to DNF*. December 2003. 11 pp. A preliminary version appeared in Rovan and Vojtás, editors, *Mathematical Foundations of Computer Science: 28th International Symposium*, MFCS '03 Proceedings, LNCS 2747, 2003, pages 612–621.

RS-03-44 Anna Gál and Peter Bro Miltersen. *The Cell Probe Complexity of Succinct Data Structures*. December 2003. 17 pp. An early version of this paper appeared in Baeten, Lenstra, Parrow and Woeginger, editors, *30th International Colloquium on Automata, Languages, and Programming*, ICALP '03 Proceedings, LNCS 2719, 2003, pages 332–344.

RS-03-43 Mikkel Nygaard and Glynn Winskel. *Domain Theory for Concurrency*. December 2003. 45 pp. To appear in a *Theoretical Computer Science* special issue on Domain Theory.

RS-03-42 Mikkel Nygaard and Glynn Winskel. *Full Abstraction for HOPLA*. December 2003. 25 pp. Appears in Amadio and Lugiez, editors, *Concurrency Theory: 14th International Conference*, CONCUR '03 Proceedings, LNCS 2761, 2003, pages 383–398.

RS-03-41 Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. *An Operational Foundation for Delimited Continuations*. December 2003. 21 pp.