Guided Controller Synthesis for Climate Controller Using Uppaal Tiga

Jan J. Jessen[1], Jacob I. Rasmussen[2], Kim G. Larsen[2], and Alexandre David[2]

[1] Automation and Control, Aalborg University, Denmark
jjj@control.aau.dk
[2] Department of Computer Science, Aalborg University, Denmark
illum@cs.aau.dk

Abstract. We present a complete tool chain for automatic controller synthesis using Uppaal Tiga and Simulink. The tool chain is explored using an industrial case study for climate control in a pig stable. The problem is modeled as a game, and we use Uppaal Tiga to automatically synthesize safe strategies that are transformed for input to Simulink, which is used to run simulations on the controller and generate code that can be executed in an actual pig stable provided by industrial partner Skov A/S. The model allows for guiding the synthesis process and generate different strategies that are compared through simulations.

1 Introduction

Inevitable parts in a traditional control design cycle are modelling, simulations and synthesis. Modelling often results in non-linear continuous models needing linearization and/or model order reduction in order to be applicable for control, while simulation can implement both original and linearized models. For control synthesis standard linear controllers are verified by design, but the control engineer still needs to perform the step of translating a mathematical description of the controller into an executable application that can be run on an embedded platform. Additionally, in the setting of hybrid models controller synthesis itself is a highly non-trivial task.

In this paper, we present a prototype for model-based framework for optimal control using the recently developed controller synthesis tool Uppaal Tiga [3,2] in combination with Simulink [10] and Real-Time Workshop [12] providing a complete tool chain for modeling, synthesis, simulation and automatic generation of production code (see Fig. 1). The framework requires that two models of the control problem are provided: An abstract model in terms of a timed game and a complete, dynamic model in terms of a (non-linear) hybrid system. Given the abstract (timed game) model together with logically formulated control and guiding objectives, Uppaal Tiga automatically synthesizes a strategy which is directly compiled into an S-function[1] representation of the controller. Now

[1] S-function is a term used in Simulink for executable content that can be embedded into Simulink components. S-functions support multiple languages such as C and Matlab.

J.-F. Raskin and P.S. Thiagarajan (Eds.): FORMATS 2007, LNCS 4763, pp. 227–240, 2007.
© Springer-Verlag Berlin Heidelberg 2007

using Simulink together with the concrete (dynamic) model, simulation results for additional quantitative aspects of the synthesized controller can be obtained. Alternatively, given interface code for the specific actuators and sensors, Real-Time Workshop allows for the generation of production code implementing the synthesized controller. The glue used to tie UPPAAL TIGA together with Simulink has been hand-coded for the purpose of this paper. For an of the shelf tool chain, this glue need to be implemented into UPPAAL TIGA making S-functions an output format.

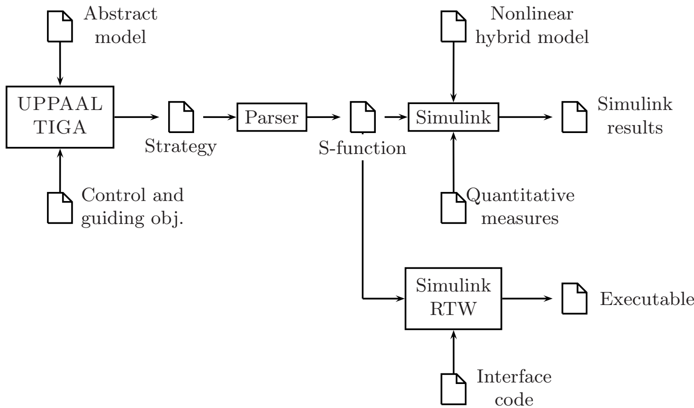


Fig. 1. Illustration of tool chain for model based control

The framework is presented through an industrial case study carried out in collaboration with the company Skov A/S specializing in climate control systems used for modern intensive animal production. For such systems it is of extreme importance that the climate control work properly, since a failure can result in the death of entire batches of animals and in turn loss of revenue for the farmer. In this context a *properly* functioning control system should additionally provide a comfortable environment for the animals.

In [9,8], a dynamic model for a pig stable that is both nonlinear and hybrid and a verified stable temperature controller has been presented. The control design of said papers is unique and does not apply standard control design techniques. We show in this paper that our framework allows for automatic generation of the controller presented in [9,8], and moreover that our framework makes it straight forward to obtain and implement extended controllers, e.g. by including humidity control. For a thorough discussion of the control engineering issues with controller synthesis for the climate controller, we refer to [9,8].

The model of the climate controller is constructed in an ad-hoc manner and the paper does not provide methodology for abstracting non-linear hybrid models to timed automata models. The model serves two different purposes, namely, to illustrate the tool chain for deriving production code from models and provide the area of automatic controller synthesis of real-time systems with an industrial scale case study. Furthermore, the constructed model does not include clock

variables, hence, it does not need the features timed games. However, the climate problem needs a real-time controller and other models using clocks could be constructed, and for that we reason present the work in terms of timed automata. Also, the advanced modelling features of UPPAAL TIGA (e.g. functions and selections) make it an attractive choice for modelling games, even if these do not use clock variables.

In Section 2 we describe a dynamic, zone-based climate model for the evolution of temperature in a pig stable. In Section 3 we briefly describe UPPAAL TIGA together with the notions of timed game, control objective and strategies. Section 4 is the main section giving a detailed description of how the climate controller is modelled and synthesized with UPPAAL TIGA. Numerical results are presented in Section 5, and conclusions are given in Section 6.

2 Climate Model

In this section, we introduce the dynamic climate model describing the evolution of temperature in a pig stable. The presented model is zone based, a concept where the pig stable is divided into distinct climatic zones, and where the zones interact by exchanging air flow. The idea is illustrated in Fig. 2 where a stable is partitioned into N subareas, and where the zones exchange air flow.

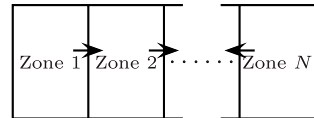


Fig. 2. N zones

Though it would be relevant to model temperature, humidity, CO2 and ammonia concentration we initially limit ourselves to modeling only temperature, in order to illustrate the zone concept. It would though be easy to include the disregarded climate parameters since the mixing dynamics are, roughly, identical.

Assumption 1. *Climatic interdependence between zones is assumed solely through internal air flow.*

With assumption 1 we thus neglect radiation and diffusion etc. between zones, claiming they are negligible compared to the effect from having internal air flow. Besides internal air flow a zone interact with the ambient environment by activating a ventilator in an exhaust pipe and consequently opening a screen to let fresh air into the building. Air flowing from outside into the ith zone is denoted Q_i^{in} [m^3/s], from inside to outside Q_i^{fan} [m^3/s]. Air flowing from zone i to $i+1$ is denoted

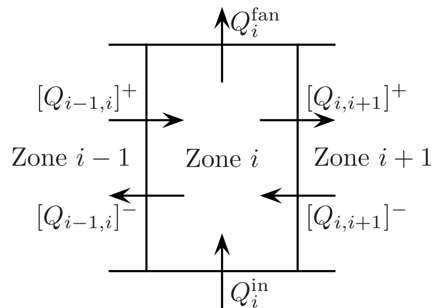


Fig. 3. Illustration of flows for zone i

$Q_{i,i+1}$ [m^3/s] (air flow is defined positive from a lower index to a higher index). A stationary flow balance for each zone i is found:

$$Q_{i-1,i} + Q_i^{\text{in}} = Q_{i,i+1} + Q_i^{\text{fan}} \quad (1)$$

where by definition $Q_{0,1} = Q_{N,N+1} = 0$.

The flow balance is illustrated in Fig. 3 using the following definitions: $[x]^+ \triangleq \max(0,x)$, $[x]^- \triangleq \min(0,x)$. In accordance with [4,1] and taking into account the flows leaving/entering the ith zone, the following model for temperature evolution is easily obtained.

$$\begin{aligned} \frac{dT_i}{dt}V_i = T^{\text{amb}}Q_i^{\text{in}} - T_i Q_i^{\text{fan}} + [Q_{i-1,i}]^+ T_{i-1} - [Q_{i-1,i}]^- T_i \\ - [Q_{i,i+1}]^+ T_i + [Q_{i,i+1}]^- T_{i+1} + \frac{u_i^t + W_i^t}{\rho_{\text{air}} c_{\text{air}}} \end{aligned} \quad (2)$$

where V_i [m^3] is the zone volume, T^{amb} [$^\circ$C] is the ambient temperature, Q_i^{in}, Q_i^{fan} is the inflow and outflow respectively. c_{air} [J/(kg$^\circ$C)] is the specific heat capacity of air, ρ_{air} [kg/m^3] is the air density. u_i^t [J/s] is the controlled heating and W_i^t [J/s] is heat production from the pigs. For the actuator signals maximum values exists $Q_i^{\text{fan}} \in [0, Q_i^{\text{fan},M}]$, $Q_i^{\text{in}} \in [0, Q_i^{\text{in},M}]$, $u_i^t \in [0, u_i^{t,M}]$. The disturbance is not known but bounded $W_i^t \in [W_i^{t,m}, W_i^{t,M}]$.

In [9] a temperature controller for the model in (2) is presented. The presented controller is a multi-zone controller, i.e., it consists of N individual (yet identical) controllers. The controller is event-based, and only changes its control action when certain boundaries are met or a neighboring zone changes its control action. The controller in [9] is designed to solve a two player game theoretic problem following [7] at each time a state has changed or a change in coordinating variables take place. Each controller maintains a set of coordinating variables δ^i that holds information about the controllers willingness to exchange air flow with the neighboring zones, and only if two neighboring zones agree to the exchange, air will flow between the zones. The game theoretic view of the control problem for an arbitrary zone enables the same generated controller to be implemented in all zones of a N-zone stable. The correctness of this is explicitly proved in [9].

The control actions available to controller is the heating u_i^t, opening of the inlets Q_i^{in} and turning on the ventilators Q_i^{in}. The controller has two "modes" heating up and cooling down, and an initial mode set to either one. We remark specifically that opening of the inlet is not enough to force air into the zone. This being a physical system air has to be removed either by operating the ventilator or by having a neighboring zone extract air. The controller operation in zone i is as follows: When heating up ventilation is closed and heating is turned on. If in addition a neighbor zone has warmer air than in the current zone, the controller will inform the neighboring zone's controller that it would like to receive the warmer air. Only if the two zones agree to exchange air will the controller in zone i turn on its ventilation fan extracting warm air from one of the neighbor zones. When cooling down, the heating is turned off, the inlets opened and

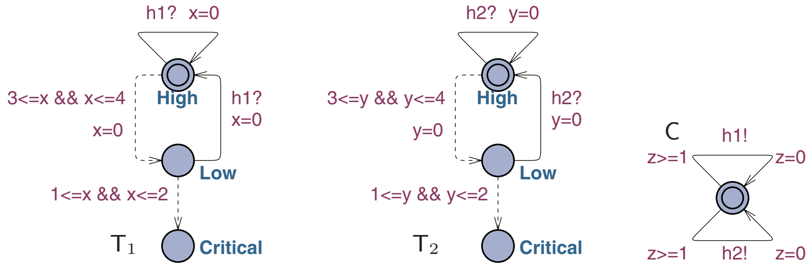


Fig. 4. Two Tank Temperature Control Problem

the ventilation fan is turned on. The controller will in addition inform the two neighbor zones, that it would like to "give away" air thus forcing more fresh air into the zone.

3 Timed Games, Control Objectives and Strategies

UPPAAL TIGA is a tool for solving control problems modeled as (networks of) timed game automata [3]. As an example consider the control problem in Fig. 4, where a central controller C is to maintain the temperature of two tanks, T_1 and T_2 above some critical minimum level, say 5°C. Each tank is modelled as a timed game automaton with location `High` indicating that the temperature in the tank is between 80°C and 100°C. Similarly, the `Low` locations indicate a temperature between 10°C and 15°C and the `Critical` locations that temperature is below 5°C. The controller C has the possibility for heating either tank thus lifting (or maintaining) its temperature to the `High` level; the act of heating is modelled as synchronizations on the channels $\mathtt{h}_1$ and $\mathtt{h}_2$. The guards $\mathtt{z} \geq 1$ on the clock `z` of the controller enforces that heating actions of C are seperated by at least 1 time-unit. The dashed edges in the two tanks represent *uncontrollable* transitions for lowering the temperature (from `High` to `Low` and from `Low` to `Critical`) in a tank in case no heating action of the controller has taken place for a certain time period; e.g. the guard $3 \leq \mathtt{x} \wedge \mathtt{x} \leq 4$ indicates that the temperature in T_1 may drop from `High` to `Low` at any moment between 3 and 4 time-units since the last heating of the tank.

Control purposes are formulated as "`control: P`", where `P` is a TCTL formula specifying either a safety property, (`A[]` φ) or a liveness property (`A<>` φ). Given a control purpose, "`control: P`", the search engine of UPPAAL TIGA will provide a *strategy* (if any such exists) for the controller under which the behaviour of the model will satisfy `P`. Here a strategy is a function that in any given state of the game informs the controller what to do either in terms of "performing a controllable action" or to "delay". In our tank example of Fig. 4 the control purpose may be formulated as "`control: A[] not(`T_1`.Critical or `T_2`.Critical)`". Endeed there is a strategy guaranteeing the safety property involved (i.e., the `Critical` temperature level is avoided in both tanks). In the case when the two

tanks are both having a `Low` temperature level the strategy provided by UPPAAL TIGA requests the controller to heat T_2 whenever $(2 < \mathtt{y} \wedge 1 < \mathtt{z} \wedge \mathtt{y} \leq \mathtt{x}) \vee ((2 < \mathtt{x} \wedge 1 < \mathtt{z}) \wedge (\mathtt{y} < 1 \vee \mathtt{x} < \mathtt{y}))$. In case $(2 < \mathtt{y} \wedge 1 < \mathtt{z} \wedge \mathtt{x} < 1)$ the strategy suggest to heat T_1. Interestingly, it may be shown (as discovered by UPPAAL TIGA), that for slower controllers (e.g. replacing the guards $\mathtt{z} \geq 1$ by $\mathtt{z} \geq 2$) no strategy exists which will ensure our control purpose.

UPPAAL TIGA is integrated in the UPPAAL 4.0 framework permitting the use of discrete (shared or global) variables over simple or structured types (arrays and recods) including user-defined types. Functions can be declared using C-like syntax and used in guards and update statements. Edges have an additional select statement as a shorthand notation for all edges that satisfy the statement.

4 Modelling

In this section, we give a detailed description of the adhoc method in which the climate controller has been modelled in UPPAAL TIGA. We divide the description into a model section and a property section with guiding. Note that some of the UPPAAL TIGA code snipplets could be given a mathematical description; we have chosen the code in order to allow for the precise reconstrution of our method.

4.1 The Models

The compound model consists of three kinds of automata, the neighbor automaton, an auxilliary automaton, and controller automaton. Each of these are described in turn in the following.

Neighbor Automaton. The neighbor model is an automaton with just uncontrollable transitions that can change the observable variables of the neighboring zone. The template for the neighbor automaton is depicted in Fig. 5 and is instantiated with a parameter `id` which can take the values 0 and 1 to indicate the left and right neighbor.

Each neighbor has a variable `temp` that discretizes the temperature information of the neighbor to either `HOTTER` or `COLDER` than the control zone. Furthermore, there is a variable `n` that holds the values of the interaction variables of the neighbor. The variable `n`, which can take any of the values `WANT`, `HAVE` and `NEITHER` (encoded as the type `choice_t`), is used to indicate whether the neighbor wants air flow from the control zone, wants to deliver air flow to the control zone, or does not want to exchange air flow with the control zone.

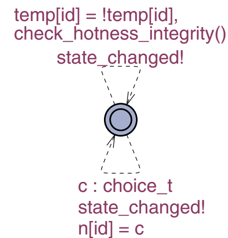


Fig. 5. Neighbor Automaton

To switch the temperature of a neighboring zone, the environment can take the uncontrollable transition at the top of Fig. 5. The call to the fucntion `check_hotness_integrity()` on the transition is explained below. The bottom

transition uses special UPPAAL TIGA syntax for select statements. This is shorthand notation for the three cases where `c` takes on any of the values of `choice_t`, i.e., the environment can set the control variables of the neighbor to any kind of desired interaction. Whenever the environment changes an observable variable it synchronizes over the channel `state_changed` with the controller, to allow the controller to change the control strategy. This way we keep a strictly alternating game where the controller reacts every time an observable variable changes value.

Auxiliary Automaton. To manage the other observable variables, we introduce an auxiliary automaton that allows the environment to change these variables. The auxiliary automaton is depicted in Fig. 6.

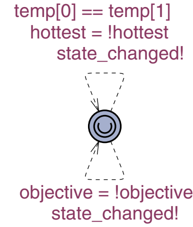


Fig. 6. Auxiliary Automaton

The final two observable variables that can change are, first, the variable `objective` which determine whether the control zone should `HEATUP` or `COOLDOWN` (bottom transition of the automaton). The second variable is a result of the discretization of the temperature information. The control zone needs information about which neighbor has hotter air. This is encoded using the Boolean observable variable `hottest` where value 0 indicates the left neighbor is hotter and vice versa for value 1. The environment can change the value of `hottest` on the top transition only when either both zones are either colder or hotter, otherwise the value can become inconsistent with the temperatures of the neighbors. The function call `check_hotness_integrity` is used by the neighbor automaton whenever the temperature changes to guarantee that `hottest` is left in a consistent state.

Controller Automaton. The controller automaton synchronizes with the auxiliary automaton and neighbor automata over the channel `state_changed` whenever an observable variable changes values. Upon synchronization, the controller enters the committed state `Decide` and the setting of the control variables is determined on the transition exiting `Decide`. The controller automaton is depicted in Fig. 7

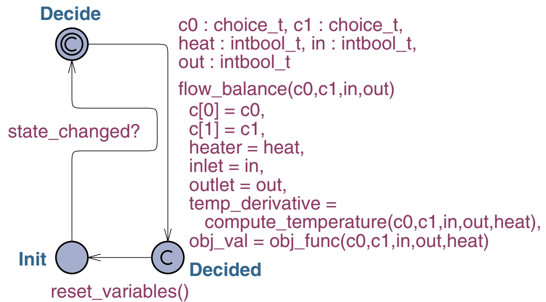


Fig. 7. Controller Automaton

The controller automaton determines the value of five control variables: Two variables for the interaction with the neighbors (`c[0]` and `c[1]`) and one variable for each of the `heater`, `inlet` and `outlet` (the latter three are all Boolean variables). The selection statement on the transition from `Decide` to `Decided` guarantees that all possible settings are considered. The guard statement

`flow_balance` guarantees that no inconsistent control state wrt. air flow is considered, i.e., whenever air is flowing out of a zone, air is flowing into the zone (see Algorithm 1) and vice versa. After updating the control variables the combined impact of the control variables and the observable variables on the temperature of the control zone is computed using the function `compute_temperature`. We refer to `obj_func` later, when we talk about guiding.

Upon entering the committed location `Decided` the transition back to `Init` is taken immediately which resets all the control variables. This is merely a step to minimize the state space since, as we shall see, the effect of the control decision is only important in `Decided`.

Algorithm 1. Procedure to guarantee that the flow balance is satisfied.

```
proc flow_balance(c0,c1,in,out) : bool
 1: bool o = out || (n[0]==WANT && c0==HAVE) || (n[1]==WANT && c1==HAVE)
 2: bool i =  in || (n[0]==HAVE && c0==WANT) || (n[1]==HAVE && c1==WANT)
 3: return o == i
```

Discretization of the Temperature Derivative. Since the model is discretized such that the controller does not know the exact temperature of the neighboring zones, this needs to be reflected in the computed temperature derivative.

We choose to let the different control parameters contribute to the temperature derivative according to the table to the right. The values for the airflows correspond to opening the outlet fan and getting air only from the specific source. Given that the fan capacities are fixed, getting air from multiple sources will share the capacity. E.g., getting air from both (hotter) neighbors would yield a contribution of 1 from the hottest and 0.5 from the coldest, resulting in a total contribution of 1.5. Furthermore, having multiple sources of outflow increases the inflow contribution proportionally, e.g., allowing the inlet to give a total contribution of -21 by opening the outlet and providing air for both neighbors.

Heater:	5
Inlet:	-7
Hotter neighbor	
- hottest:	2
- coldest:	1
Colder neighbor	
- hottest:	1
- coldest:	2

Computing the Temperature Derivative. As we saw above, the fans have a fixed capacity that might be shared among the different sources of outflow. Since this can result in a non-integral contribution and Uppaal Tiga only handles integers, we need to multiply these contributions with an apropriate factor to guarantee integral values. Since a single source of outflow can be shared among up to three sources of inflow, we choose a constant `OUT_CONTRIBUTION=6` to denote the available contribution per outflow source as this can be integrally shared among the potential inflow sources. This has the added effect that we need to multiply the heater contribution by six as well, to keep the proportions.

The function for computing the temperature derivative is listed in Algorithm 2. Lines 1 and 2 compute the contributors to air flow in and out of the

zone. For outflow, this, in order, corresponds to 1) is the outlet open, 2) is air flowing from the control zone to the left zone, and 3) similarly for the right zone. The computation is analogous for air flowing into the control zone.

The value of amp computed in line 3 is the contribution for each inflow given the total outflow. Now, the return statement computes the total effect of the control decision by using the table above and the amplifier for each inflow contribution. Note that the heat contribution is also amplified to keep the the proportions defined above.

The final two negative parts of the contribution are used to indicate that giving air away cools the zone. These are used as incentives to let the controller offer air when it wants to cool. The reason is that when the controller is used in all zones we can imagine the situation when one zone needs to cool and a neighbor want the air to heat up. In the control situation when neither are interacting, one of the zones need to intiate the cooperation, and this is accomplished with the given incentives. Note that these values are negligible in the overall contribution.

Algorithm 2. Algorithm for computing the temperature derivative.

```
proc compute_temperature(c0,c1,in,out,heat) : int
 1: int outflow = out + (c0==HAVE && n[0]==WANT)+(c1==HAVE && n[1]==WANT)
 2: int inflow = in + (c0==WANT && n[0]==HAVE)+(c1==WANT && n[1]==HAVE)
 3: int amp = (outflow * OUT_CONTRIBUTION) / inflow
 4: return OUT_CONTRIBUTION*5*heat
    + amp*(c0==WANT && n[0]==HAVE ? (temp[0]? (!hottest? 2:1) : ( hottest? -2:-1)) : 0)
    + amp*(c1==WANT && n[1]==HAVE ? (temp[1]? ( hottest? 2:1) : (!hottest? -2:-1)) : 0)
    + amp*(in ? -7 : 0)
    - (c0==HAVE) - (c1==HAVE)
```

4.2 The Property

In order to synthesize the controller, we need to specify the property that the resulting controller should synthesize. An immediate choice would be[2]:

$$\phi \equiv \texttt{control : A[] Controller.Decided imply (objective ? 1 : -1)*temp_derivative > 0} \quad (3)$$

In other words, invariantly whenever the controller enters Decided, the value of temp_derivative should be greater than zero when heating is the objective and less than zero when the objective is cooling. However, this property would be satisfied by the simple controller that never interacts with the neighbors and turn on the heater when the objective is heating and opens the inlet and outlet when the objective is cooling.

Guiding. With the property above we can determine whether we can satisfy the main objective or not. Now, we define an objective function called obj_func that will guide the controller synthesis process while also satisfying the property

[2] Recall that we switched the sign of the temperature derivative when the objective is to cool down.

above[3]. Given an appropriate objective function, the following property can be used to guide the controller synthesis process[4]:

```
φ ≡ control : A[] ZC.Decided imply forall (c0 : choice_t) forall (c1 : choice_t)
    forall (in : intbool_t) forall (out : intbool_t) forall (heat : intbool_t)
    flow_balance(c0,c1,in,out) imply obj_val >= obj_func(c0,c1,in,out,heat)
```

In plain words, the property states that it should hold invariantly that whenever the controller makes a decision and enters the location `Decided`, then for all other possible controller choices that satisfy the flow balance, the computed objective function is smaller or equal to the choice made. In short, the controller always chooses a configuration of the control variables that maximizes `obj_func` among all valid choices.

The simplest objective function is to use `compute_temperature`, but to compensate for the sign depending of the objective as (3) above. This guiding proccess will produce a controller that maximizes (minimizes) the temperature derivative for every control decision. An alternate strategy is to define the objective function over some sort of energy consumption by, e.g., penelizing turning on the heater or fan, thus, optimizing towards energy optimality.

4.3 Controlling Humidity

As mentioned in [9], the climate controller should, ideally, be extended with the ability to control the humidity in the stable as well. However, the approach outlined in [9] makes this extension a tedious strategy, since the increase in oberservable variables creates exponentially more configurations.

Changing our model to accommodate for humidity as well, requires a slight modification to the models along the lines of how the temperature was modelled. Furthermore, the objective function needs to represent the effect of temperature and humidity with a simple value. Note, that neither the controller automaton nor the property changes, as the set of controllable variables remains unchanged.

To discretize the humidity readings of the neighboring zones, analogously to the temperature representation, we introduce a Boolean `humid` variable for each zone and a `morehumid` variable to determine which of the neighbors have air with the highest humidity. Obviously, there are constraints on consistent variable assignments for the three variables in the same way as for the temperature variables.

To incorporate the variables in the model, we need an extra uncontrollable transition in the neighbor automaton, that can change the value of the respective `humid` variable. This is followed by a consistency check on the `morehumid` variable. Moreover, we add two extra uncontrollable transitions to the auxiliary automaton, one to change the `more_humid` variable, and one to change the objective

[3] To satisfy both properties we use conjunction, but do not include the conjuction to simplify properties.

[4] Note we use guiding in the sense that the function determines which of a number of valid controllers to choose, but the synthesis process itself is not guided in order to find controllers.

with respect to humidity which is encoded in the variable `decrease_humidity`. These are all the changes needed to the automata.

We incorporate humidity information in the objective function in a similar fashion the temperature model with the exception that humidity only has an upper limit, so the objective is either to decrease the humidity or ignore the humidity. Thus, when `decrease_humidity` has value false, the humidity contributes nothing to the objective function. Otherwise, the contribution is given as in Algorithm 3 where a postive value indicates a decrease in humidity. In the algorithm, `amp` is computed as for the temperature contribution. The positive contributions are from opening the inlet (contribution of 5) and getting less humid air (contribution of 2 for the least humid air and 1 for the most humid). Receiving more humid air from a neighboring zones contributes negatively. Finally, we encourage a zone to interact, if a neighboring zone has less humid air, even if the zone does not want to interact (first two parts of the sum).

Algorithm 3. Humidity contribution to the objective function

```
1: return (!humid[0] && c0 == WANT ? 1 : 0) + (!humid[1] && c1 == WANT ? 1 : 0)
   + amp*(c0==WANT && n[0]==HAVE ? (humid[0] ? (!morehumid?-2:-1):( morehumid?2:1)) :0)
   + amp*(c1==WANT && n[1]==HAVE ? (humid[1] ? ( morehumid?-2:-1):(!morehumid?2:1)) :0)
   + amp*(5*in)
```

The objective function is constructed with a weight parameter between the temperature derivative and the humidity parameter with changing the sign of the temperature as above. The weighing can be altered to generate different controllers, which later can be compared in some appropriate fashion as discussed in Section 5.

5 Results

In this section, we present some numerical results where the controller generated by Uppaal Tiga has been simulated in Simulink using realistic values for the model (2).

The Tool Chain. According to Fig. 1, to generate production code for the climate controller of the pig stable, we need to transform the output format of Uppaal Tiga to input for Simulink. Simulink allows input of so-called S-functions which are user provided C-code that can be used within the Simulink model. We have build a script which takes Uppaal Tiga strategies as input and delivers S-functions as output. The Simulink model with the S-function can be used to either run simulations of the pig stable, or generate Comedi compliant production code through Real-Time Workshop. The code generation is realized through a Comedi library for Simulink [5]. Code generation with Real-Time Workshop allows for a multitude of targets, thus, the specific target of this application in unimportant.

Numerical Results. We have synthesized two types of controllers using the models described above. One controlling only the temperature, and one controlling both temperature and humidity. In the first experiment, we synthesized a controller for temperature only as explained in Section 4. Due to limited space, we choose not to include the graphs for the experiments, as the synthesized controller is identical to that of, [9,8]. As in [9,8], the controller behaves well under simulation and keeps the zone temperatures within the given bounds.

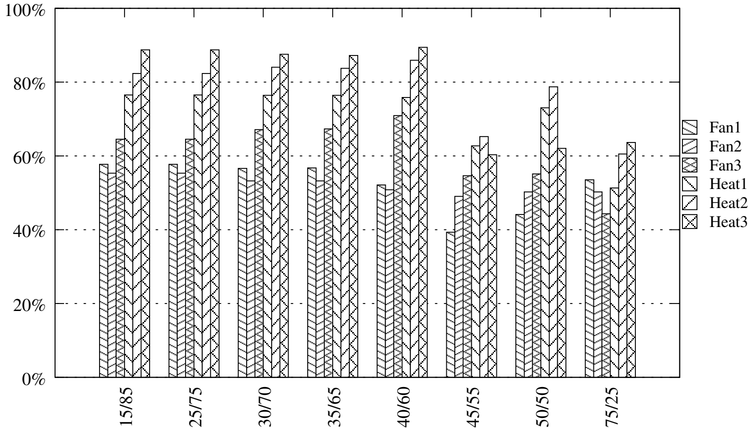
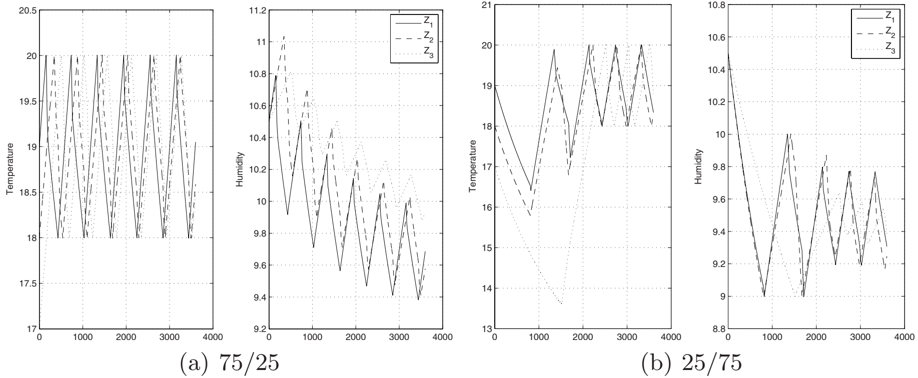


Fig. 8. Active time for heaters and fans for different controllers. The number (x/y) indicate an objective function using x percent temerature contribution and y percent humidity contribution.

In order to illustrate the guiding specification in Uppaal Tiga a number of different controllers are simulated in Simulink. A weight is put on the objective function guiding towards temperature or humidity control. The simulation scenario is as follows: The stable is partitioned into 3 zones, and the thermal boundary is set to [18 20] and for humidity [9 10] for all three zones. The initial conditions are set to $T_1 = 19$, $T_2 = 18$ and $T_3 = 17$, $H_1 = H_2 = H_3 = 11$. All the conducted experiments stear the state to the defined boundaries in finite time, but initially some states are steared away from the boundary. In order to quantify and compare the different controllers the total time when the heat or fan are on is recorded. The result is illustrated in Fig. 8. The results show, that the controllers can be divided up into two catagories, one from 0% to 40% temperature guided, and one from 45% to 100%. The controllers in the latter category use less heat and fan capacity than the controllers in the former catagory, indicating that the former are preferred controllers. However, Fig. 9 shows how the temperature and humidity are controlled for controllers in both catagories. As it can be seen, the controller with more heat and fan activation (25/75) reaches a stable state faster than the controller with less activity of heaters and fans.[5] Thus, the

[5] Simulation results for all controllers can be found at the project website, [11].



(a) 75/25 (b) 25/75

Fig. 9. Simulation results for temperature and humidity when guiding towards a) 75% temperature and 25% humidity and b) vice versa

choice between the controllers is not immediately clear, but the quatifications can be used by the control engineers to make an informed choice.

Note that we have not tested the code in the actual pig stable. However, for the temperature controller we can rely on the results provided in [6], the the generated controller is identical to that of [9,8]. These results show that the controller does not have identical quantitative properties to the simulation, though, the qualitative properties are identical. I.e. the experiments show that the temperature oscillates as in the simulation, however, the temperature in the real stable under/overshoots the limits. This is mainly caused by the fact that the zones are not thermically isolated in the sense that air will interchange between zones, even when the zones do not what to interchange air.

For the humidity controller we do not have any experimental data to rely on, but this will be investigated in the future.

6 Conclusions and Future Work

In this paper, we have presented a complete tool chain for automatic controller synthesis from timed game automata models to production code. For the livestock production case study, the controller synthesis process has enabled, through guiding, to synthesize an identical controller do that of [9,8]. The controller in [9,8] was synthesized in a tedious manual way, which indicates the importance of a simple automated process. Note that the notion of time was not necessary in modelling our controller, however, we choose UPPAAL TIGA because the tool was available and one the only ones of it's kind.

Furthermore, the model was easily extended to include humidity, which was left as a matter to explore in [9,8], but never pursued due to the heavy time requirement of the added exponential complexity. With an appropriately defined weighted objective function, UPPAAL TIGA was used to synthesize a controller

capable of regulating temperature as well as humidity in a matter of seconds. A number of controllers were synthesized with variying weights between temperature and humidity, and all were able to reach stable temperature and humidity conditions in Simulink simulations. Simulink was further used to track the heat and fan activity for the different controllers, in order to allow for comparison of different controllers. This can be a very effective strategy for differentiating controllers and choosing an appropriate one among a number of controllers satisfying the conditions. As future work, we want to continue conducting experiments in the real life pig stable provided by Skov A/S in order to evaluate the different controllers capacity of controlling temperature as well as humidity in a real life setting.

References

1. Arvanitis, K.G., Soldatos, A.G., Daskalov, P.I., Pasgianos, G.D., Sigrimis, N.A.: Nonlinear robust temperature-humidity control in livestock buidings. Submitted to Biosystems Engineering (2003)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, D., Lime, K.G.: Uppaal tiga: Time for playing games. In: proc. of Computer Aided Verification (CAV'07) (to appear, 2007)
3. Uppaal Tiga Homepage (2006), http://www.cs.aau.dk/~adavid/tiga
4. Janssens, K., Van Brecht, A., Zerihun Desta, T., Boonen, C., Berckmans, D.: Modeling the internal dynamics of energy and mass transfer in an imperfectly mixed ventilated airspace. Indoor Air 14, 146–153 (2004)
5. Jessen, J.J., Schiøler, H., Nielsen, J.F.D., Jensen, M.R.: Cots technologies for integrating development environment, remote monitoring and control of livestock stable climate. In: Proceedings 2006 IEEE International Conference on Systems, Man, and Cybernetics (2006)
6. Jessen, J.J.: Embedded Controller Design for Pig Stable Ventilation Systems. PhD thesis, Automation and Control, Department of Electronic Systems, Aalborg University (2007)
7. Lygeros, J., Tomlin, C., Sastry, S.: Controllers for reachability specifications for hybrid systems. Automatica 35, 349–370 (1999)
8. De Persis, C., Jessen, J.J., Izadi-Zamanabadi, R., Schioler, H.: A distributed control algorithm for internal flow management in a multi-zone climate unit. International Journal of Control. Accepted for publication (to appear)
9. De Persis, C., Jessen, J.J., Izadi-Zamanabadi, R., Schioler, H.: Internal flow management in a multi-zone climate control unit. In: Invited paper in the session Networked Control Systems 2006 CCA/CACSD/ISIC (2006)
10. SIMULINK (2007), http://www.mathworks.com/products/simulink/
11. Project Web (January 2007), http://www.cs.aau.dk/~illum/automatic_control/
12. Real-Time Workshop (2007), http://www.mathworks.com/products/rtw/