Application of Model-Checking Technology to Controller Synthesis*

Alexandre David[1], Jacob Deleuran Grunnet[2], Jan Jakob Jessen[1], Kim Guldstrand Larsen[1], Jacob Illum Rasmussen[3]

[1] Department of Computer Science, Aalborg University, Denmark (email: {adavid,kgl,jjjessen}@cs.aau.dk)
[2] LAC engineering, Hinnerup, Denmark (email: jag@lacengineering.com)
[3] Sanddru R&D, Nørresundby, Denmark (email: illum@sanddru.com)

Abstract. In this paper we present two frameworks that have been implemented to link traditional model-checking techniques to the domain of control. The techniques are based on solving a timed game and using the resulting solution (a strategy) as a controller. The obtained discrete controller must fit within its continuous environment, which is modelled and taken care of in our frameworks. Our first technique does it by using *Matlab* to discretise the problem and then UPPAAL-TIGA to solve the obtained timed game. This is implemented as a toolbox. The second technique relies on the user defining a timed game model in UPPAAL-TIGA. Then the strategy is automatically imported in *Simulink* as an S-function for simulation and validation purposes. We demonstrate the effectiveness of these frameworks in different case-studies.

1 Introduction

The traditional control design cycle includes modelling, simulation, equation solving, and implementation. Modelling the environment and physical systems often means having to deal with non-linear or even hybrid models (mixing both discrete and continuous aspects) for which many of the standard control design methods are not easily applicable.

A major task for any control system designer is abstracting such models in to a form which is suitable for controller design e.g. by linearisation and when the controller design has been performed to simulations to validate the approximation and implementation of the control strategy. This is non-trivial and in this article we report on two prototypes for model-based design for optimal control using the controller synthesis tool UPPAAL-TIGA, *Matlab*, and its powerful toolbox *Simulink* [13].

The ultimate goal is to automate and unify the entire procedure such that the control system designer can perform modelling, synthesis and verification in a single tool, while providing only the system specification and requirements. The first prototype is the toolbox called *PAHSCTRL* [6] that enables computation of

* Work supported by the MULTIFORM project FP7-ICT-2007-2.

piecewise-affine control laws for hybrid systems with non-deterministic discrete transitions. In particular, this can be used for fault-tolerant control [8]. The second prototype is a generalisation of the climate controller case-study [11] implemented in the form of Ruby scripts that are called from within *Matlab* to integrate seamlessly with *Simulink*.

In this paper we gather previous results obtained for both *PAHSCTRL* and UPPAAL-TIGA. In addition, we define a more general framework for linking UPPAAL-TIGA to *Simulink* and detail its implementation. We first give the background of timed games, then we present *PAHSCTRL*, and finally the framework for linking UPPAAL-TIGA to *Simulink*.

2 Controller Synthesis with Timed Game Automata

In our setting we use the model of timed game automata, an extension of timed automata, to define a game between two players: a controller and an environment. The goal is find a strategy for the controller player to meet a control objective for any move of the environment player. We refer to [2, 5] for more details on the formalism, here we only summarise the important notions.

Let X be a finite set of real-valued variables called clocks. We note $\mathcal{C}(X)$ the set of constraints φ generated by the grammar: $\varphi ::= x \sim k \mid x - y \sim k \mid \varphi \wedge \varphi$ where $k \in \mathbb{Z}$, $x, y \in X$ and $\sim \in \{<, \leq, =, >, \geq\}$. $\mathcal{B}(X)$ is the subset of $\mathcal{C}(X)$ that uses only rectangular constraints of the form $x \sim k$.

Definition 1. *A* Timed Automaton *(TA) [1] is a tuple* $A = (L, l_0, \Sigma, X, E, Inv)$ *where* L *is a finite set of* locations, $l_0 \in L$ *is the* initial *location,* Σ *is the set of* actions, X *is a finite set of* real-valued clocks, $Inv : L \to \mathcal{B}(X)$ *associates to each location its* invariant *and* $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *is a finite set of* transitions, *where* $t = (l, g, a, R, l') \in E$ *represents a transition from the location* l *to* l', *labelled by* a, *with the guard* g, *that resets the clocks in* R. *One special label* τ *is used to code the fact that a transition is not observable.*

Definition 2. *A* Timed Game Automaton (TGA) *[12] is a timed automaton* G *with its set of transitions* E *partitioned into* controllable *(*E^c*) and* uncontrollable *(*E^u*) actions. In addition, invariants are restricted to* $Inv : L \to \mathcal{B}'(X)$ *where* $\mathcal{B}'$ *is the subset of* $\mathcal{B}$ *using constraints of the form* $x \leq k$.

Given a TGA G and a control property $\phi \equiv \mathcal{A}\ \phi_1\ \mathcal{U}\ \phi_2$ (*resp.* $\mathcal{A}\ \phi_1\ \mathcal{W}\ \phi_2$) of ATCTL, the *reachability (*resp. *safety) control problem* consists in finding a *strategy* f for the controller such that all the runs of G supervised by f satisfy the formula[4]. A strategy is a mapping from states to action to perform, an action being just to delay or to delay some time and to take a transition. The controller synthesis problem is formulated in our setting as a timed game to solve and the resulting controller is the strategy obtained. We refer to strategies being winning when there is a strategy for the controller player to meet its control objective.

[4] Here $\mathcal{U}$ stands for the until operator and $\mathcal{W}$ for the weak until operator. In the context of UPPAAL-TIGA, they have slightly different semantics than the usual operator in the sense that ϕ_1 should still be satisfied when reaching ϕ_2.

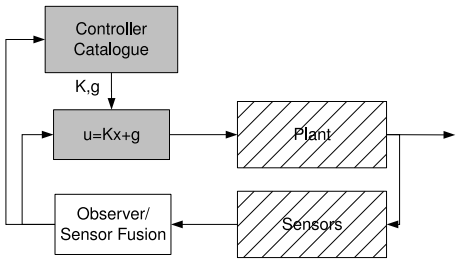


Fig. 1: The *PAHSCTRL* controller structure. Grey boxes are synthesised, the white box represents the part of the control system not handled and hatched boxes represent the physical plant and sensor systems.

3 *PAHSCTRL*

3.1 Introduction

The focus of this toolbox and framework is to automatically generate controllers for piecewise-affine hybrid systems (PAHS). Using the proposed method, fault tolerant controllers are designed by modelling faults as uncontrollable events causing switches between discrete modes. The design method involves abstracting PAHS to discrete games and deriving controllers based on winning strategies for the game.

The idea of solving control problems by abstracting to discrete or timed games is not in itself new. The inspiration to use control to ensure that the system behaviour conforms to the discrete abstraction comes from [14], which demonstrates how controllers for discrete linear systems can be designed to conform to Linear Temporal Logics (LTL)-specifications.

Our method builds on advances in controller synthesis for affine systems on polytopes [10, 9], where it is also suggested to design controllers for PAHS by abstraction. The toolbox presented here and detailed in [8] expands on these ideas, principally by adding (uncontrollable) external events, which can trigger transitions between modes.

The result is a *Matlab* toolbox capable of computing discrete game abstractions of PAHS and deriving control laws based on solutions to the discrete game. This enables computation of a type of *gain scheduling controller* where the control gains are adjusted based on the position in the state-space and the current fault condition. The resulting controller structure is shown in figure 1.

The toolbox implements and optimises the algorithms shown in [7] that details how a discrete game abstraction can be obtained for non-deterministic piecewise-affine hybrid system (PAHS). The game may be solved automatically with respect to reachability properties using Uppaal-tiga. Assuming that a winning strategy exists for this game, it can be interpreted as a rule base that determines which control law to use in a given condition, and m-functions refine the result to affine control laws, thus synthesising the complete control system.

Our goal is to automate the procedure of finding control laws for hybrid systems with non-deterministic discrete state transitions. This is achieved by calculating a catalogue of affine control laws each acting on a subset of the state-space. Together they should guarantee a set of control requirements in the form of reach/avoid specifications.

In this framework, the user formalises the hybrid system and enters it inside *Matlab*. Then the *PAHSCTRL* tool generates the discrete game, whereas in the second framework the user has to make this model.

The toolbox is implemented in *Matlab* and consists of a number of m-functions designed to be easy to use but still exposing enough functionality that the toolbox can be used for different purposes. Compared to the algorithms presented in [7], a number of optimisations have been implemented along with new functionalities, in particular regarding refinement of control laws. The toolbox is detailed in [6] and can be found at `http://pahsctrl.polytekniker.dk`.

We demonstrate which kind of problems can be solved and we outline the solution strategy employed.

3.2 Problem Definition

Informally a PAHS as defined in *PAHSCTRL* is a discrete automaton where each location is associated with a continuous system. The locations are referred to as *modes* and the state space of each mode is partitioned into polytopes each associated with an affine system of the form

$$\dot{x} = Ax + Bu + C \tag{1}$$

where x is the state variable, u is the input and A,B,C are matrices of appropriate size.

The transitions between the modes in the discrete automata can be both controllable and uncontrollable meaning that they are taken by, respectively, the controller or the environment. Such transitions can be used to model faults or other externally triggered events that change the system dynamics.

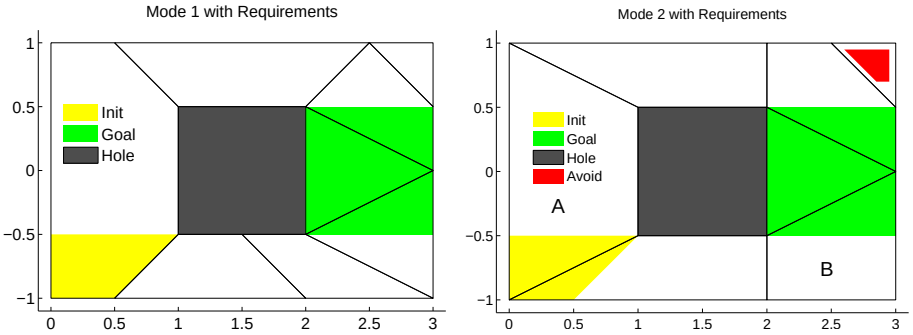
The goal of the controller synthesis is therefore to compute a control strategy that ensures that a subset of the state space of one or more modes is reached while avoiding other subsets.

This is best illustrated with the example shown in figure 2. Not shown is the uncontrollable transition from mode 1 to 2 with an identity reset map. Mode 1 can be thought of as the nominal mode while mode 2 corresponds to a fault mode.

The "hole" in the partition is a subset of state space where no dynamics have been defined. This can be used as an alternative method for describing avoid sets, with the important difference that no dynamics are specified for the "hole" and no computational effort is spent on this region of state space.

The objective of the example is to reach the goal set while staying in the partitioned set and avoiding the avoid set.

The main algorithm consists of:



(a) Partitioning of the state space in mode 1

(b) Partitioning of the state space in mode 2

Fig. 2: Partitioning of the two modes including requirements specification given as polytope sets. There is an uncontrolled transition from mode 1 to mode 2, which can occur at any point in time.

1. computing a discrete game abstraction of the hybrid system,
2. finding a solution to the game ensuring the control requirements, and
3. refining the solution to control laws of the form $u = Kx + g$.

This toolbox aims at solving 1) and 3) using *Matlab*, while leaving step 2) to established tools such as UPPAAL-TIGA.

3.3 Abstraction

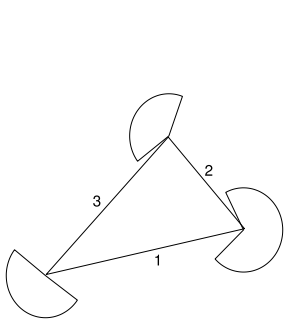
During the abstraction, discrete equivalents of each polytope are computed. That is for each polytope defined on each mode one or more abstractions are computed to encode the possible actions of an affine controller as a discrete game.

The actions are encoded according to the ability of the controller to prevent the system from leaving the polytope through a given facet. The actions are labelled as follows.

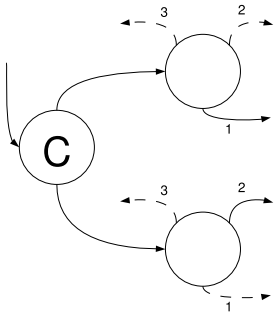
Blockable A control law exists that prevents exit through this facet
Uncontrollable The facet is not blockable.
Controllable The facet is blockable and a control law exists that can unblock the facet while ensuring that the polytope is left in finite time.

An example is presented in figure 3, showing a polytope with the controllable system directions indicated at the vertices. Dashed transitions are uncontrollable.

The discrete equivalents of a polytope are computed by solving for feasibility of linear matrix inequalities (LMI) based on the controllability to facet results presented in [9]. Each possible combination of facet labels corresponds to one LMI.



(a) A polytope where facet 3 is unblockable and *either* facet 1 or 2 can be blocked leaving the other unblockable



(b) The resulting discrete equivalent has actually 3 locations. The committed location (indicated by 'C') means that the controller must choose and the two normal locations represent the two choices.

Fig. 3: A 2-dimensional polytope being converted to a discrete equivalent in a discrete game. In this case there are two possibilities which are merged via a committed location.

3.4 Strategy

By computing discrete equivalents for all the polytopes of the PAHS and combining them, a discrete game abstraction can be obtained. In figure 4 the discrete abstraction for mode 1 of the example is shown.

With this simple example it is easy to find a winning strategy manually. The goal is to find a path from the start location to the goal location that avoids locations from where there exists a sequence of uncontrollable transitions to the hole location.

The toolbox uses Uppaal-tiga to find a winning strategy to the discrete abstraction, enabling a fully automated control synthesis.

3.5 Refinement

The affine control laws on the original PAHS are found by refining the winning strategy computed by Uppaal-tiga. The strategy determines which discrete equivalent is to be used and thus determines which LMI is used to limit the control law.

From the abstraction step it is known that the LMI chosen for each polytope is feasible and the refinement step is thus restricted to finding an optimal solution to each LMI. Combining these LMI solutions yields a controller catalogue, one controller for each polytope, which ensures that the state will reach the goal

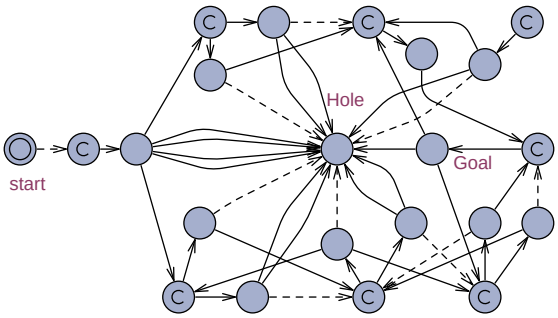
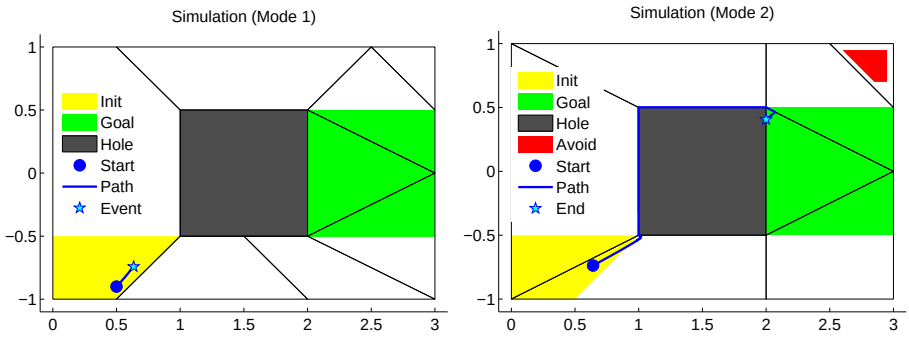


Fig. 4: A discrete game abstraction of mode 1. Each committed location is at the approximate spot of its corresponding polytope. The hole location denotes unpartitioned space.

set in finite time. A simulation of the example using a controller generated by *PAHSCTRL* is shown in figure 5. The path goes around the hole on the border (left and then above).



(a) Simulation of the controlled system starting in mode 1

(b) After the event the system continues evolution in mode 2

Fig. 5: Simulation of the example with the synthesised controller. The state ends at a fixed point just inside the goal set.

4 Linking Uppaal-tiga to *Simulink*

4.1 Introduction

Our second framework provides an integrated and complete tool chain for modelling, synthesis, simulation, and automatic generation of executable code. 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 synthesises a strategy which is directly compiled into an S-function[5].

Figure 6 shows an overview of the framework. It is based on our previous case-study of a climate controller for poultry and pig farms [11]. In this previous work, the humidity and heat transfer were described by their differential equations between zones in the farm. Then we simplified and discretised the model as a timed game automaton that was used to generate a controller automatically. That controller was then plugged into *Simulink* to validate through simulation using the non-linear model that the controller was able to control the climate as expected. It was possible to study its performance and, by choosing different control objectives, we could easily change the controller and simulate the new versions. The code generation was made possible through the *Simulink* real-time workbench. From the point where we have an S-function, we can simulate and generate real code. We have successfully redone this case-study [11] using our

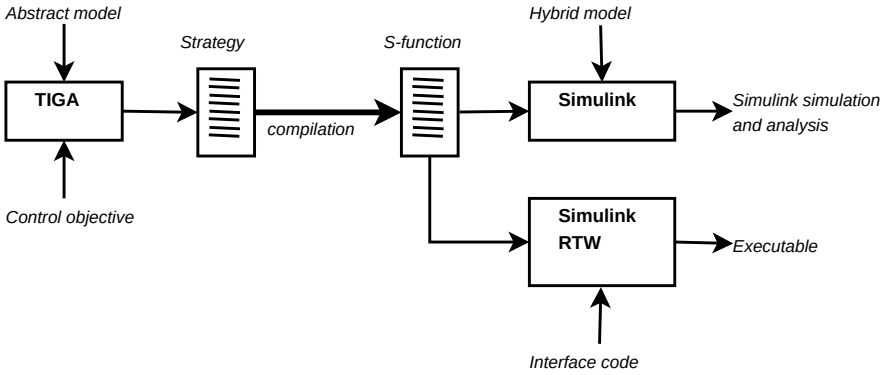


Fig. 6: Overview of the framework.

general framework instead of the custom translations. We recall that a controller was manually made taking into account only the temperature (not the humidity) and was found to be the same as the one generated by UPPAAL-TIGA. Then humidity was added to the model but this was too complex for the manual step. In addition, the objective function is given with weights on the temperature and humidity to optimise the criteria we want. This allows the generation of a series of controllers to simulate and validate their behaviours in *Simulink*. The goals

[5] S-function is a term used in *Simulink* for executable content that can be embedded into its block components. S-functions support multiple languages such as C and *Matlab* representation of the controller.

of our extensions her are to i) integrate UPPAAL-TIGA and *Simulink*, and ii) to generalise the framework.

4.2 Work-flow

In this framework the user formalises the environment and the physics of the system using classical differential equations. This is then abstracted in terms of timed game automata. As in [4] the continuous domain is discretised into intervals that correspond to clock constraints to model the dynamics. The abstract (discretised) model is entered in UPPAAL-TIGA. The model gives the possible moves for the environment and the controller players. The tool solves the game and generates a strategy (if possible) to meet a given control objective.

In parallel, the continuous model is entered in *Simulink* with a place-holder S-function that will act as the controller. Inputs and outputs for this block correspond to the UPPAAL-TIGA model. Using our translator we plug the generated discrete controller into *Simulink* to simulate it in its continuous environment. We note that it is now easy to change parameters in the model, generate new controllers and study their performances. In addition, using the *Simulink* real-time workbench allows us to generate real code for a given target platform.

4.3 Tool Integration

Figure 7 shows a more detailed view of the tool integration that we have implemented. The implementation is separated into one (internal) *Matlab* function that acts as the coordinator component and a Ruby script that makes the translation from a strategy to an S-function. The user defines a timed game automaton model in UPPAAL-TIGA together with a *Simulink* model that contains a block in which the user wishes to insert the generated controller from UPPAAL-TIGA. It is up to the user to define the input and the output variables. These inputs and outputs are defined in *Simulink* and their names must match the corresponding variables in the UPPAAL-TIGA model. The user should make sure that the desired property is satisfied to obtain a strategy. Then the user calls the *Matlab* function that

1. calls UPPAAL-TIGA to generate the strategy,
2. extracts the inputs and outputs from the *Simulink* model and generates input and output files,
3. calls the Ruby script that translates the strategy together with the declaration files of inputs and outputs into an S-function,
4. and calls the *Matlab* C-compiler to compile the generated S-function and imports the binary into *Simulink*.

The *Simulink* model can now be simulated with the generated controller or it can be used with the real-time workbench to generate code from the S-function.

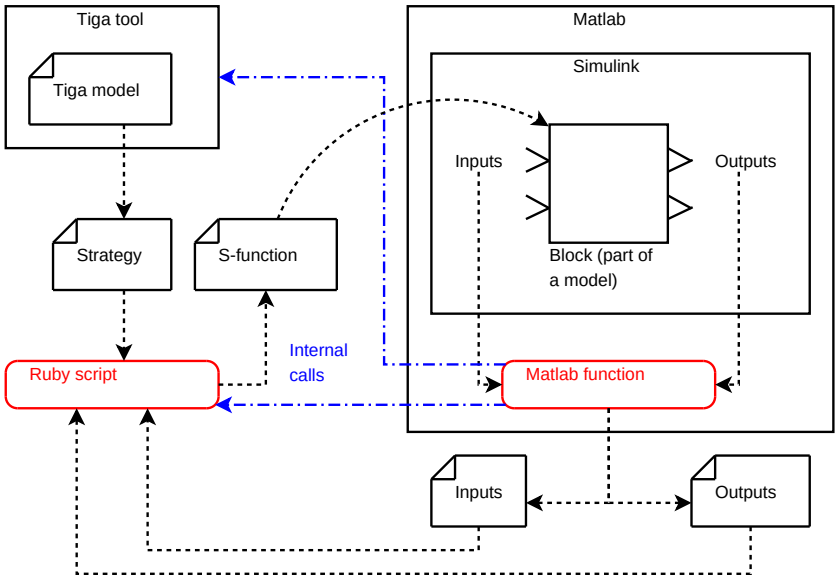


Fig. 7: Integration of UPPAAL-TIGA and *Simulink*.

4.4 Mapping to *Simulink*

To have the generated strategy (from UPPAAL-TIGA) work in *Simulink*, the models need to obey a few constraints. First *Simulink* will play the uncontrollable transitions but they should not change location in our model, only integer variables. This models the input from the environment. In our example we define that the temperature variables (in fact the indices) are the inputs. They are allowed to change according to our model in UPPAAL-TIGA and the model in *Simulink* should match this behaviour.

Second, we need to define outputs from our controller to *Simulink*. The controller can change its own locations, variables, and clock. We define that some of these variables are used as output to *Simulink*.

Finally, time is discretised by *Simulink* with some resolution. Our strategy is ultimately transformed to an S-function that is in fact a decision function with some added code to make the interface between *Simulink* variables and state variables of the controller. Clocks are incremented at every call of the function and the strategy decides what to do at every *tick* (possibly just wait).

The transformation from our strategy (mapping from states to action) is done as if-statements that transform the updates in the timed game automata into statements. Furthermore, if we had used functions in the model, they are evaluated and transformed into simple assignment statements, which results in a strategy devoid of functions in UPPAAL-TIGA syntax. This is possible because such functions have their output solely determined by the discrete state they are evaluated on and states are known in the strategies. The generated code

starts by accessing the input and output ports of *Simulink*. Incrementing the clocks is then done after taking the actions to prepare for the next call of the function. The trade-off in this solution is that we let the user test clock values for zero upon the first call but afterwards we will never get zeros again since the discretisation forces a minimal time between the action and the next time we can read inputs and take a decision again. The user will be able to simulate the generated strategy and see if the system is stable with the chosen parameters in spite of the discretisation.

4.5 Methodology and Example

The first task is to abstract the physical model to a timed game automaton. In our extension, *timed* controllers are supported[6] and they are integrated in *Simulink* by discretising time. The abstraction here consists in mapping the continuous behaviour of a system to the time dimension and to make control decision based on chosen intervals. The goal is to keep the abstraction as coarse as possible to simplify the controller but in principle we could discretise with a fine granularity and model the behaviour as precisely as we want.

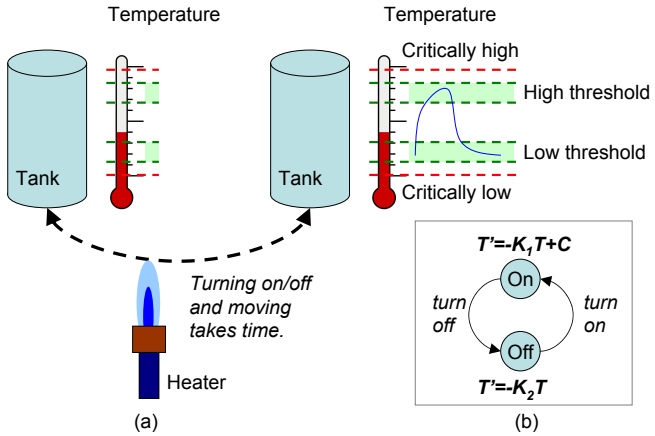


Fig. 8: The 2-tank example. One heater can heat one tank at a time and moving the heater between the tanks takes time (a). The temperature of the tanks should stay within an acceptable range. The temperature is modelled by the simple hybrid system in (b) with two states associated with differential equations.

To illustrate the modelling step, we consider a 2-tank example as shown in figure 8.(a). The idea is to maintain the temperature of two tanks containing

[6] This is in contrast to our previous work where only untimed strategies were supported by our framework.

some liquid within some specified bounds. We have one heater that can be used to heat either one of the two tanks, but changing tank takes time. The temperature of the tanks should be kept between a safe middle range and in our abstraction we consider critical low or high temperatures that we do not want to reach and two ranges of temperatures that are observable by our controller. These serve as low and high thresholds as shown in the figure. The hybrid model of the dynamics is simple here as shown in figure 8.(b). We have a state machine (for each tank) with two states to denote when the heater is on or off with associated differential equations to describe how the temperature changes. T is the temperature, K_1, K_2, and C are constants.

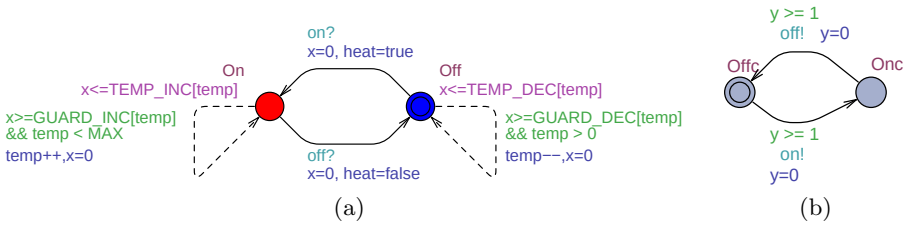


Fig. 9: The model of the 2-tank example in UPPAAL-TIGA.

We model this system in UPPAAL-TIGA with one process per tank and one for the controller. Figure 9 shows the templates for the tank and the controller. The tank automaton (Fig. 9.(a)) reflects the two states of the heater being on and off and a clock `x` is used to measure time.

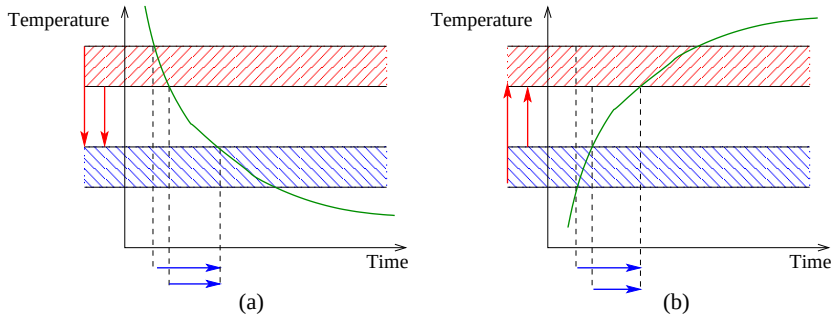


Fig. 10: Principle for mapping temperature changes to time when the temperature is decreasing (a) or increasing (b). We obtain a lower and an upper bound on time for changing temperature range.

Temperature changes are then mapped to time intervals and the model is designed to take uncertainties into account. Figure 10 shows the principle. In

Fig. 10.(a) the temperature decreases from somewhere from the high observable range to the lower one. We derive a lower and upper bound on time for detecting the state change. Similarly we derive time bounds when the temperature increases in Fig. 10.(b). The lower bounds are modelled by the guards (`x>=GUARD_DEC[temp]` and `x>=GUARD_INC[temp]` when the temperature is decreasing or increasing) and the upper bounds are the invariants (`x<=TEMP_DEC[temp]` and `x<=TEMP_INC[temp]` depending on heating). The model is designed to discretise an arbitrary number of such observable ranges and we make experiments with two and three such ranges. The controller (Fig. 9.(b)) models that it can turn a heater on or off with a constraint on time.

Given some dynamic model in *Simulink*, we extract the time ranges that we insert in UPPAAL-TIGA. We first make the experiments with the following ranges:

- Above 100, temperature is critical high (`temp`=3).
- Between 70 and 90, temperature is high (`temp`=`HIGH`=2).
- Between 40 and 60, temperature is low (`temp`=`LOW`=1).
- Below 30, temperature is critical low (`temp`=0).

The corresponding time intervals in the models are declared as follows[7]:

```
const int TEMP_INC[temperature_t] = { 0, 6, 7, 0 };
const int TEMP_DEC[temperature_t] = { 0, 18, 10, 0 };
const int GUARD_INC[temperature_t] = { 0, 2, 2, 0 };
const int GUARD_DEC[temperature_t] = { 0, 7, 3, 0 };
```

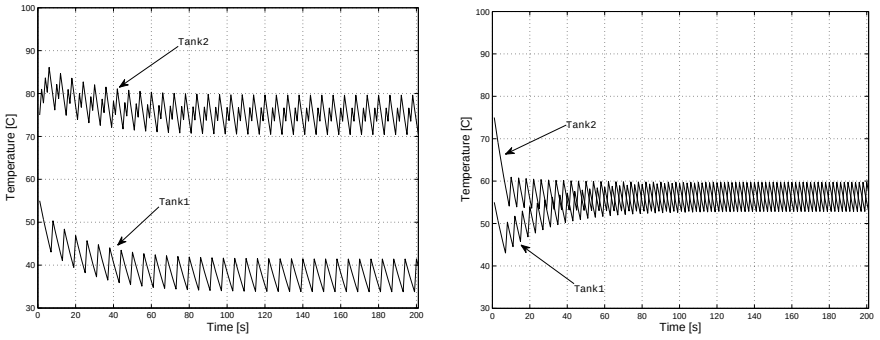
The system is initialised with tank 1 at 55 degrees and tank 2 at 75 degrees, which corresponds to `temp` being 1 and 2. We note that the model detects changes of temperature so the actual range of temperature depends on the state (heating or not). We ask for the following control objectives:

```
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=LOW  && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=LOW  && temp2>=LOW  && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=HIGH && temp2<=HIGH
```

The two first objectives are met and UPPAAL-TIGA generates strategies that we insert in *Simulink*. The third one is not due to the constraints of the model (there is no winning strategy for this game). We plot in figure 11 the result of the simulations for the first (a) and second (b) properties. We note that first, having `temp1` staying at `LOW` depends much on the timing parameters because there is no other observable range that the controller can use. Stability of the simulated system depends on the uncertainties used in the model. Second, UPPAAL-TIGA generates one arbitrary strategy that is only guaranteed to meet a control objective in the model. The simulation allows the user to evaluate its performance. With the loose specification of the first property, the controller chooses to keep one tank at a high temperature and the second one at a low temperature. The

[7] The 0 entries do not matter since we want to avoid these states. The parameters in *Simulink* are arbitrary, the important point is to derive our constants from them.

choice is natural w.r.t. their initial conditions. For the second property the controller chooses to keep both tanks in the same range even though the previous strategy could have been enough. This is not a bug in the controller since the temperatures both follow their specifications.



(a) (b)

Fig. 11: Simulation results with two observable ranges, one simulation for each control objective.

We repeat the experiments by defining the following ranges instead:

- Above 100, temperature is critical high (`temp`=4).
- Between 80 and 90, temperature is high (`temp`=`HIGH`=3).
- Between 60 and 70, temperature is good (`temp`=`GOOD`=2).
- Between 40 and 50, temperature is low (`temp`=`LOW`=1).
- Below 30, temperature is critical low (`temp`=0).

The corresponding declaration of parameters is:

```
const int TEMP_INC[temperature_t] = { 0, 4, 5, 5, 0 };
const int TEMP_DEC[temperature_t] = { 0, 13, 9, 7, 0 };
const int GUARD_INC[temperature_t] = { 0, 2, 2, 2, 0 };
const int GUARD_DEC[temperature_t] = { 0, 7, 4, 3, 0 };
```

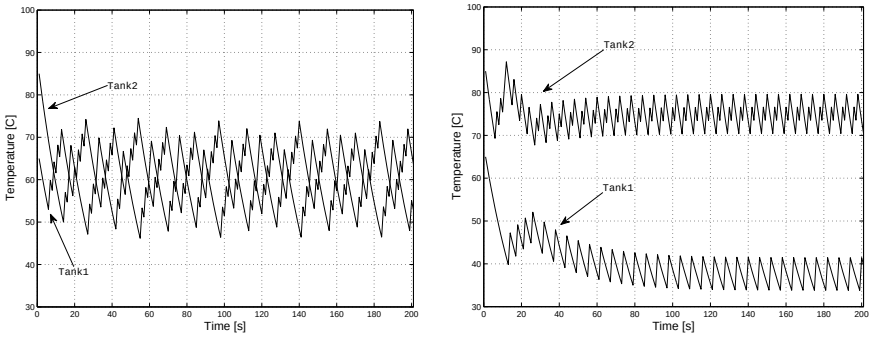
We update the initial temperatures to be 65 and 85 for the two tanks with the corresponding `temp` being 2 and 3. We check for the following control objectives:

```
control: A[] temp1>=LOW && temp1<=HIGH && temp2>=LOW  && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=GOOD && temp2>=GOOD && temp2<=HIGH
control: A[] temp1>=LOW && temp1<=GOOD && temp2>=HIGH && temp2<=HIGH
```

Similarly the two first properties are satisfied but not the third one. We show the result of the simulation in figure 12. For the first property the controller chooses to keep both tanks within the same (large) range of temperatures. The second property results in separating the temperatures, as was the intention. We also

experienced strategies in our experiments that would be similar to Fig. 11.(b) and still meet their control objectives.

The attentive reader would notice that for Fig. 11.(a) and Fig. 12.(b) the actual simulated temperature gets below 40 degrees though still above 30 degrees. The difference in the interpretation of the control objective comes from the fact that there is no temperature in the game model and the resulting controller uses the threshold "bands" as observations in a manner similar to [3] by detecting entering and leaving observations. The discretized controller takes decisions when crossing 40 degrees and never observes the temperature falling below 30 degrees. The parameters of these models would need to be refined to take decisions when crossing 50 degrees instead, which can be achieved by asking a different control objective.



(a) (b)

Fig. 12: Simulation results with three observable ranges, one simulation for each control objective.

We showed a methodology in this case-study to go from a hybrid model to a time model to generate a discrete controller. The approach matches the reality of having sensors that will detect changes (here of temperature) with some precision. The approach shows promising results.

5 Conclusion and Future Works

We have presented two frameworks that can be used to generate hybrid controllers and bridge the gap between control theory and its implementation on real hardware. Our case-studies show the viability of these approaches. Common for both methods is the use of (timed) game abstractions in order to get the problems on a computational tractable form. The *PAHSCTRL* toolbox enables automatic abstraction and refinement to and from discrete game form while the UPPAAL-TIGA-*Simulink* framework can simulate, solve and generate code for timed games.

Future works include how to merge the first approach with the second one to get the complete work-flow within *Simulink* for simulation and code generation purposes. The first framework generates models and is using only *Matlab* while the second framework takes advantage of *Simulink* but requires a manually constructed model. These approaches are complementary. In addition, UPPAAL-TIGA can represent strategies as multi-terminal decision diagrams and output them as pseudo-code in a different format. This could be used to generate more compact and efficient code.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.
3. F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.
4. F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier. Automatic synthesis of robust and optimal controllers - an industrial case study. In *HSCC*, pages 90–104, 2009.
5. T. Chatain, A. David, and K. G. Larsen. Playing games with timed games. In A. Giua, C. Mahulea, M. Silva, and J. Zaytoon, editors, *Preprints of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 238–243, 2009.
6. J. D. Grunnet, T. Bak, J. D. Bendtsen, and F. Ankersen. PAHSCTRL - a control synthesis toolbox for piecewise-affine hybrid systems. In *Proceedings of the 2009 European Control conference*. IEEE, 2009.
7. J. D. Grunnet, T. Bak, J. D. Bendtsen, and J. A. Larsen. Discrete game abstraction for fault tolerant control synthesis. In *Proceedings of IEEE CACSD'08*, 2008.
8. J. D. Grunnet, J. D. Bendtsen, and T. Bak. Automated fault tolerant control synthesis based on discrete games. In *Proceedings of the 48th IEEE Conference on Decision and Control*. IEEE, 2009.
9. L. Habets and J. H. van Schuppen. Control to facet problems for affine systems on simplices and polytopes - with applications to control of hybrid systems. In *Proc. 44th IEEE CDC*, 2005.
10. L. C. G. J. M. Habets, P. J. Collins, and J. H. van Schuppen. Reachability and control synthesis for piecewise-affine hybrid systems on simplices. *IEEE Transactions on Automatic Control*, 51:938–948, 2006.
11. J. J. Jessen, J. I. Rasmussen, K. G. Larsen, and A. David. Guided controller synthesis for climate controller using UPPAAL-TIGA. In *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, number 4763 in LNCS, pages 227–240. Springer, 2007.
12. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS*, volume 900 of *LNCS*. Springer, 1995.
13. Mathworks. Simulink, 2010.
14. P. Tabuada and G. J. Pappas. Linear time logic control of discrete-time linear systems. *IEEE Transactions on Automatic Control*, 51:1862–1877, 2006.