Bridging the gap between embedded systems and automation systems

Partha S Roop

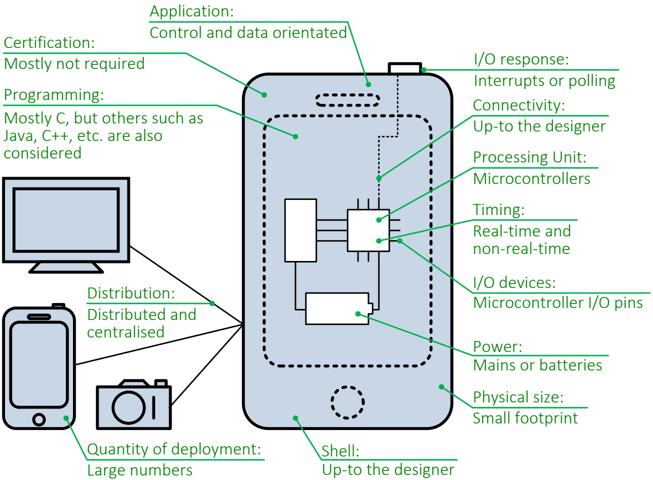
THE UNIVERSITY OF AUCKLAND NEW ZEALAND Te Whare Wānanga o Tāmaki Makaurau

BioRemulation™ Research Group
The University of Auckland

August 5 2016

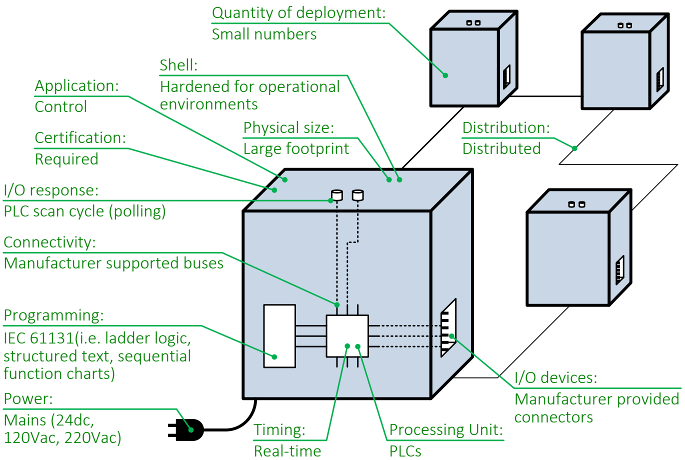
www.pretzel.ece.auckland.ac.nz/bio

Embedded Systems



(b) Embedded systems [a]

[a] Yoong et al. Model driven design using IEC61499, Springer 2015.



(a) Automation system

[a]

[a] Yoong et al. Model driven design using IEC61499, Springer 2015.

Distinction is blurring

- Real-time control using PLCs alone is not feasible any longer.
- PLCs are combined with FPGAs to bridge the gaps in timing. [a]:
- Using an FPGA allowed for encoding position signals to be handled directly from the sensors. No intermediate processing or amplification device was required, thereby reducing noise and increasing processing speed. In a process cycle faster than 1 ms, the valve position is measured and speed is calculated as both are compared to the set point. Movement is corrected using a PID algorithm. To keep the hydraulic circuit balanced, pressure values in the front and back of the cylinder are simultaneously controlled to avoid instantaneous peaks.
- Question: How to develop a systematic approach that bridges the gap without using ad-hoc solutions?

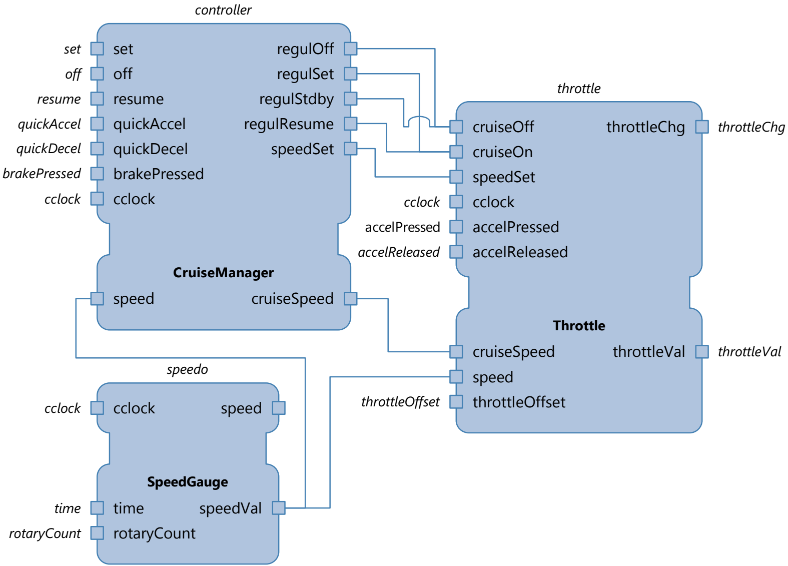
[a] Greenfiled D (2013) How embedded systems are changing automation, Automation World.

The synchronous approach

- A family of languages developed in France in the early 80s, with identical view of concurrency. [a].
- Inspired by synchronous circuits: all components trigger relative to a global clock.
- The reactive system operates *infinitely fast* relative to its environment. This is known as the synchrony hypothesis.
- Interleaving disappears in this semantics due to strict notions of *causality*.

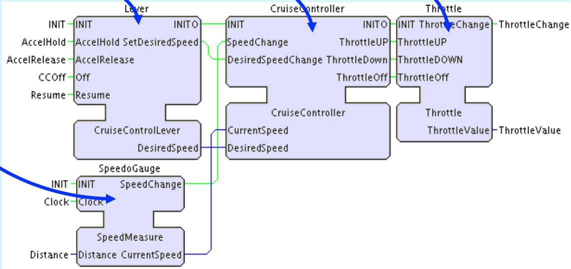
[a] A. Benveniste et al., The synchronous languages twelve years later, proceedings of IEEE, 91(1), 2003

IEC61499 – a cruise controller



Synchronous approach for IEC61499

```
module CruiseControl:
    input INIT, FootBrake, AccelHold, AccelRelease, CCOff;
    input Resume, Clock, RUN, Distance : value integer;
    output ThrottleChange, ThrottleValue : value integer;
    output SpeedChange, CurrentSpeed : value integer;
    signal Lever_SetDesiredSpeed, Lever_INITO,
           Lever_DesiredSpeed : value integer, ... in
        run Throttle [...]
    ||
        run CruiseController [...]
    ||
        run CruiseControlLever [...]
    ||
        run SpeedGauge [...]
    end signal
end module
```



4.3.1.6 Sequential statement

Rule 4.11 expresses the fact that the sequence does not finish, if its left branch, $\mathtt{t}$, does not.

$$\frac{\mathtt{t}, D \xrightarrow[I^p]{O,\perp} \mathtt{t'}, D'}{\mathtt{t;u}, D \xrightarrow[I^p]{O,\perp} \mathtt{t';u}, D'} \tag{4.11}$$

If the left branch pauses, so does the sequence.

$$\frac{\mathtt{t}, D \xrightarrow[I^p]{\emptyset,1} \mathtt{t'}, D}{\mathtt{t;u}, D \xrightarrow[I^p]{\emptyset,1} \mathtt{t';u}, D} \tag{4.12}$$

Moreover, if the left branch raises an exception (by exiting a trap), its right branch will never get executed.

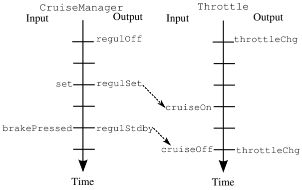
$$\frac{\mathtt{t}, D \xrightarrow[I^p]{\emptyset,k} \mathtt{t'}, D \qquad k \geq 2}{\mathtt{t;u}, D \xrightarrow[I^p]{\emptyset,k} \mathtt{t'}, D} \tag{4.13}$$

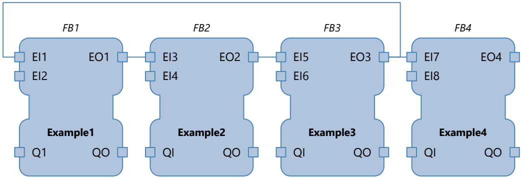
Otherwise, control will be immediately transferred to the right branch, $\mathtt{u}$, when $\mathtt{t}$ finishes.

$$\frac{\mathtt{t}, D \xrightarrow[I^p]{\emptyset,0} \mathtt{t'}, D}{\mathtt{t;u}, D \xrightarrow[I^p]{\emptyset,\perp} \mathtt{u}, D} \tag{4.14}$$

a

Delayed composition for causality





(a)

```
EI1(k) = EO3(k-1);
EI3(k) = EO1(k-1);
EI5(k) = EO2(k-1);
EI7(k) = EO3(k-1);
Compute EO1(k);
Compute EO2(k);
Compute EO3(k);
Compute EO4(k);
```

(b)

Semantics: parallel

$$\frac{\mathtt{t}, D \xrightarrow[I^p]{O,\perp} \mathtt{t'}, D'}{\mathtt{t}\|\mathtt{u}, D \xrightarrow[I^p]{O,\perp} \mathtt{t'}\|\mathtt{u}, D'} \tag{4.15}$$

$$\frac{\mathtt{u}, D \xrightarrow[I^p]{Q,\perp} \mathtt{u'}, D'}{\mathtt{t}\|\mathtt{u}, D \xrightarrow[I^p]{Q,\perp} \mathtt{t}\|\mathtt{u'}, D'} \tag{4.16}$$

Rule 4.17 uses the completion code synchronizer to specify the synchronized behaviour of the parallel statement. When both $\mathtt{t}$ and $\mathtt{u}$ perform finished transitions, the parallel statement synchronizes their execution using their completion codes.

$$\frac{\mathtt{t}, D \xrightarrow[I^p]{\emptyset,k} \mathtt{t'}, D \quad k \geq 0 \qquad \mathtt{u}, D \xrightarrow[I^p]{\emptyset,l} \mathtt{u'}, D \quad l \geq 0}{\mathtt{t}\|\mathtt{u}, D \xrightarrow[I^p]{\emptyset,syn(k,l)} \mathtt{t'}\|\mathtt{u'}, D} \tag{4.17}$$

As already mentioned in the description for data assignment, write-write concurrency on variables is disallowed, while read-write concurrency is semantically forbidden by rule 4.8. This means that $\mathtt{t}$ and $\mathtt{u}$ will never operate on the same variables in the same instant. [a]

[a] Yoong et al. A synchronous approach for IEC 61499 function block implementation, IEEE Transactions on Computers, 58(12), 2009.

Outcomes

- Determinism: Given any state and any valid input combination, at most one transition is enabled.
- Reactivity: Given any state and any valid input combination, at least one transition is enabled.
- Theorem: Synchronous function blocks are deterministic and reactive. [a]
- The unification of embedded systems and automation systems requires the consideration of cyber-physical systems i.e. not just the controller but also the plant.

[a] Yoong et al. A synchronous approach for IEC 61499 function block implementation, IEEE Transactions on Computers, 58(12), 2009.

Cyber-physical systems

Cyber-physical systems (CPS)[a] use distributed embedded controllers to control physical processes. Examples may be found in several domains: automotive, robotics, medical devices, and smart grids.

[a] R. Alur, Principles of Cyber-Physical Systems. MIT Press, 2015.

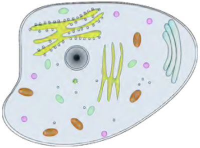
1

[1] Figure reproduced from http://icc.mtu.edu/cps/

Modelling Cyber-physical systems

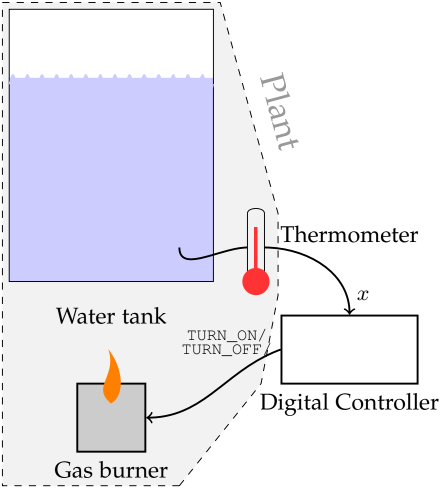
- Hybrid automata (HA) is a major enabler for the formalization of CPS.
- A combination of ODEs to model the continuous dynamics and FSMs to model the discrete mode changes that are induced by the controller.

Model: Car
Discrete: Changing gears
Continuous: Throttle control



Model: Cell biology
Discrete: External stimulus
Continuous: Flow of ions

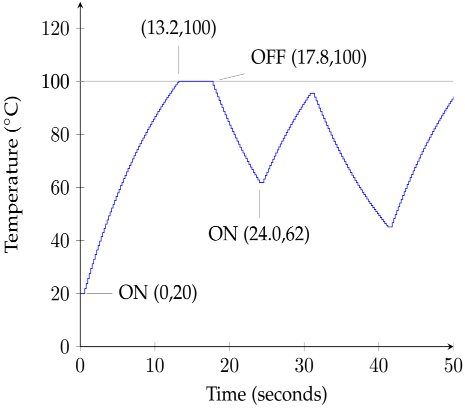
A water tank temperature controller



[a]

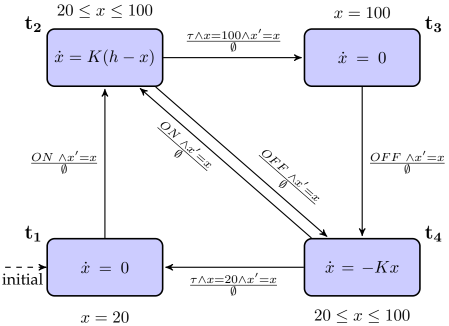
[a] J-F Raskin, "An introduction to hybrid automata", Handbook of Networked and Embedded Control Systems Control Engineering 2005, pp 491-517.

A water tank temperature controller



- Temperature of water inside a tank may be modelled as $x(t) = Ie^{-Kt} + h(1 - e^{-Kt})$ where:
- I is the initial temperature.
- K is a constant that depends on the tank conductivity.
- h is a constant that depends on the power of the gas burner.

A hybrid automata example

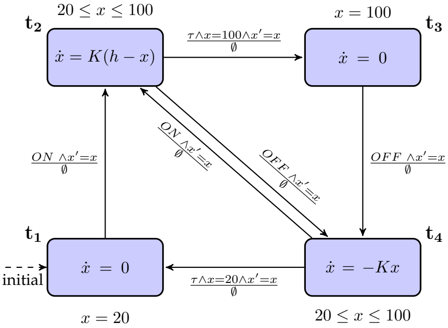


[a]

- Four locations $t1, ..t4$ that represent the discrete modes.
- Each location has some flow predicates that specify the rate of change of the continuous variables.

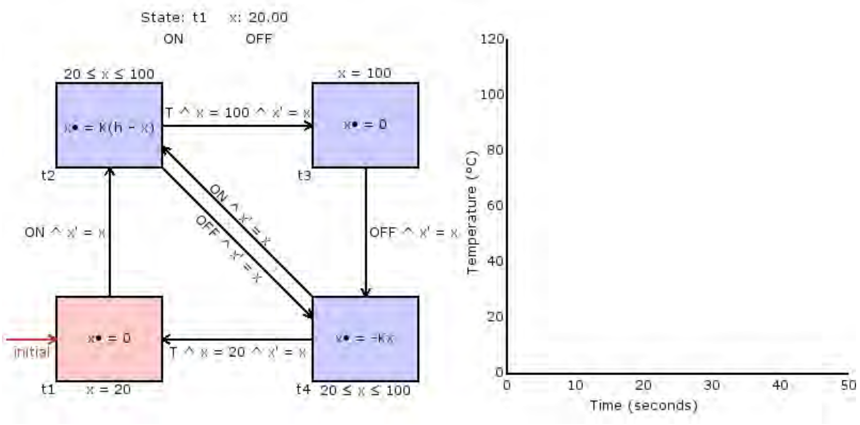
[a] J-F Raskin, "An introduction to hybrid automata", Handbook of Networked and Embedded Control Systems Control Engineering 2005, pp 491-517.

A hybrid automata example

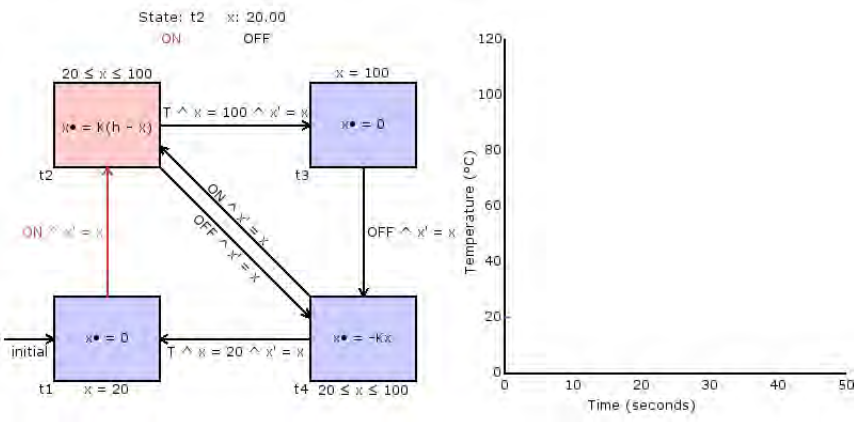


- Invariants are associated with locations e.g. $20 \le x \le 100$ is an invariant associated with $t1$. Execution remains in a location until the invariant holds.
- Some locations may have initialization conditions that provide the initial values of the variables.
- A transition is enabled when the input is present and the jump condition associated with the transition holds. When a given transition is taken the final value of the variables are updated.

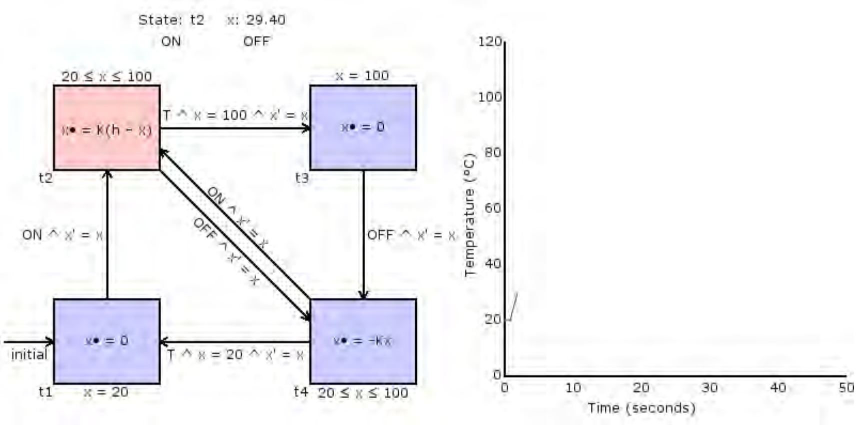
A hybrid automata example



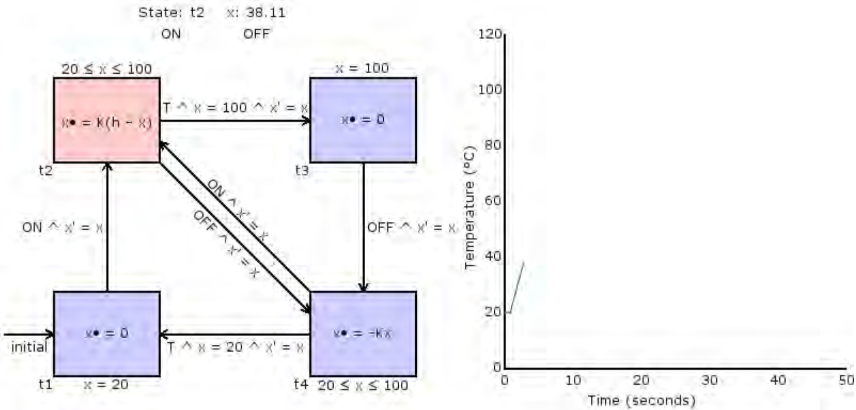
A hybrid automata example



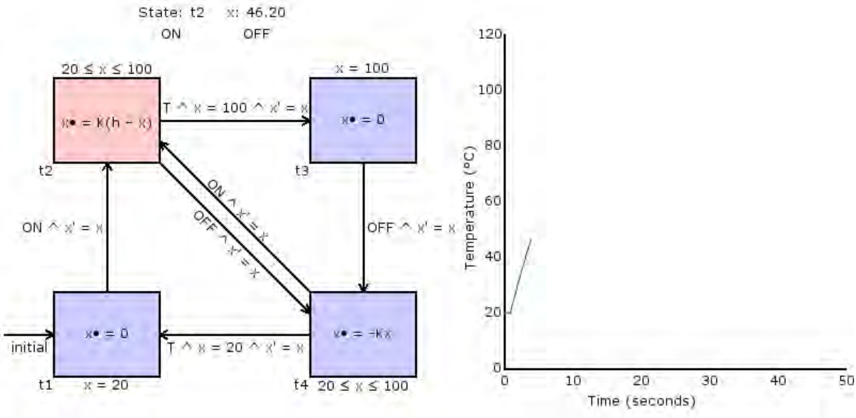
A hybrid automata example



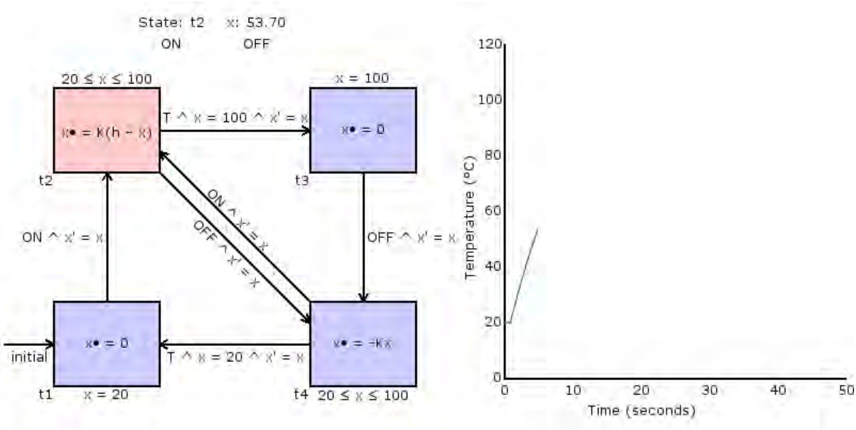
A hybrid automata example



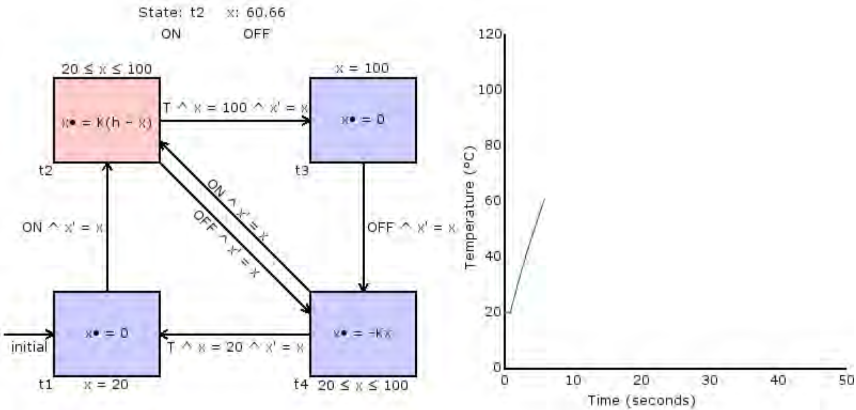
A hybrid automata example



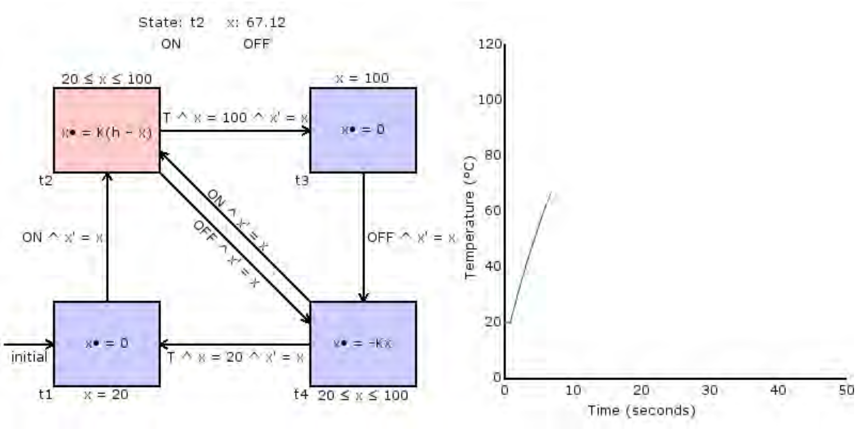
A hybrid automata example



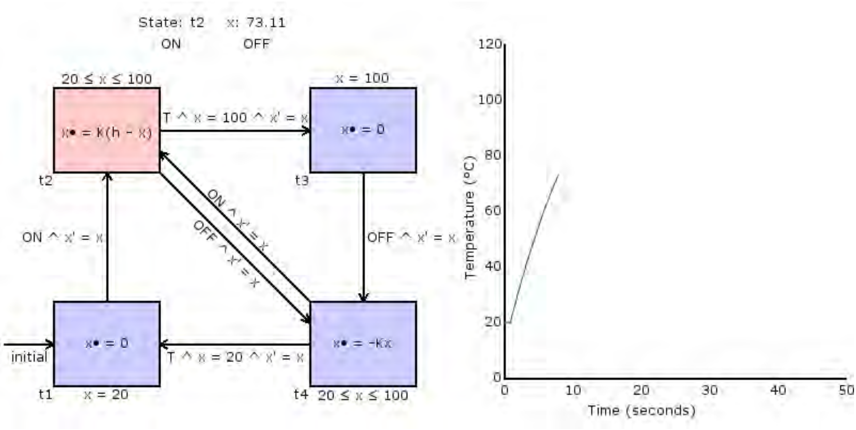
A hybrid automata example



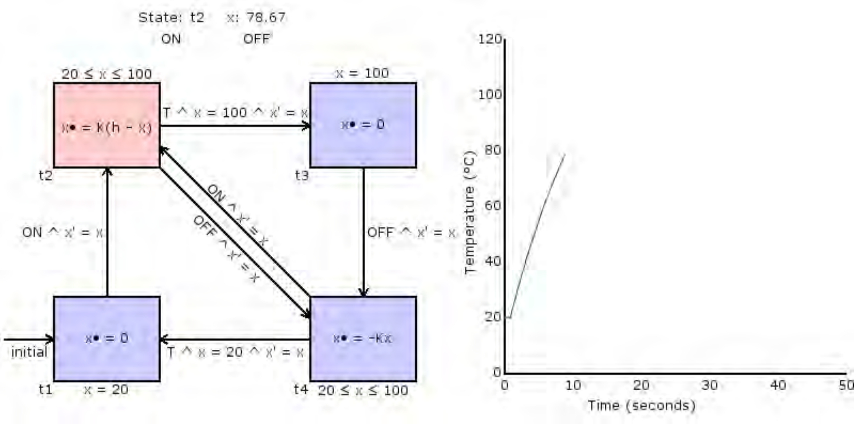
A hybrid automata example



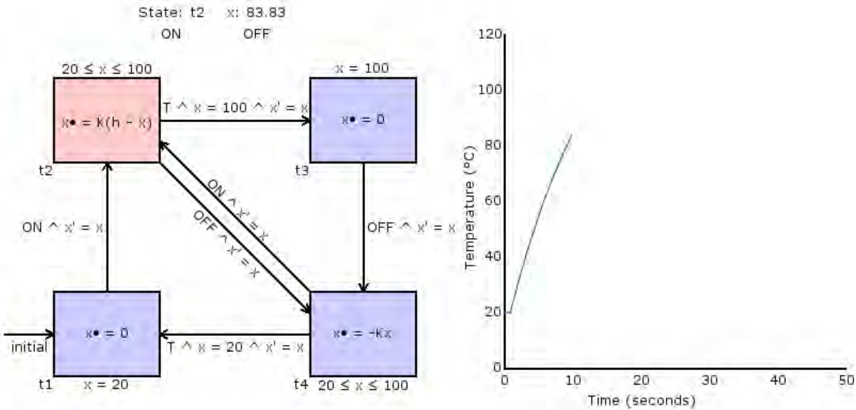
A hybrid automata example



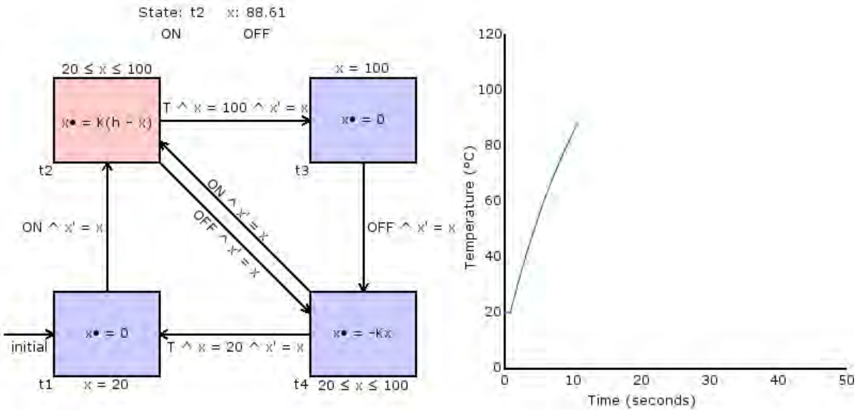
A hybrid automata example



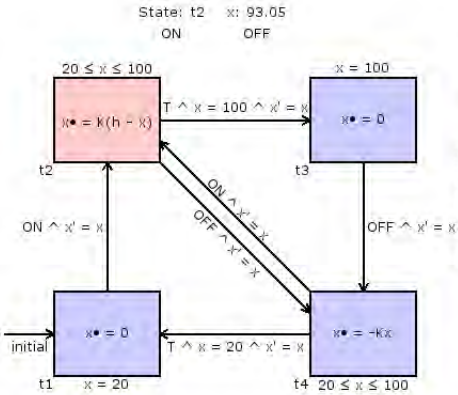
A hybrid automata example

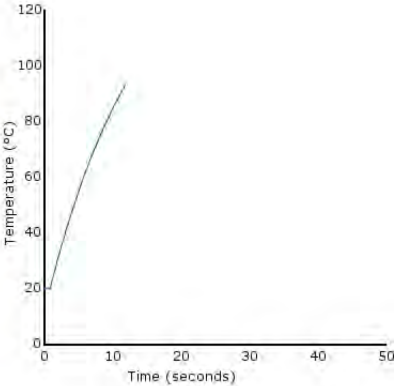


A hybrid automata example

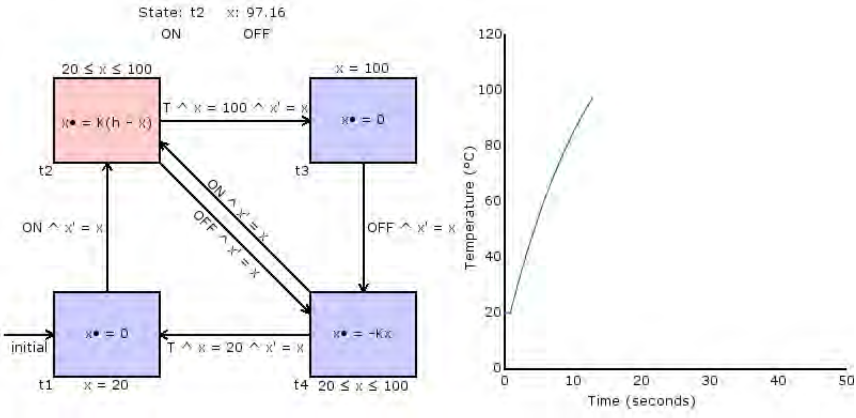


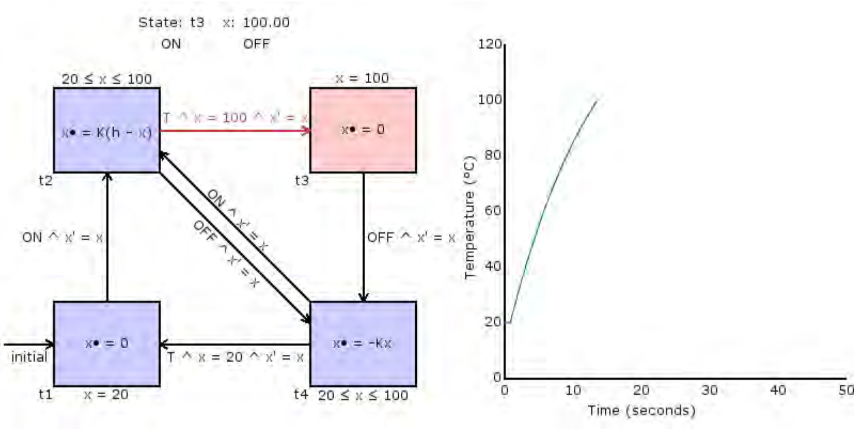
A hybrid automata example





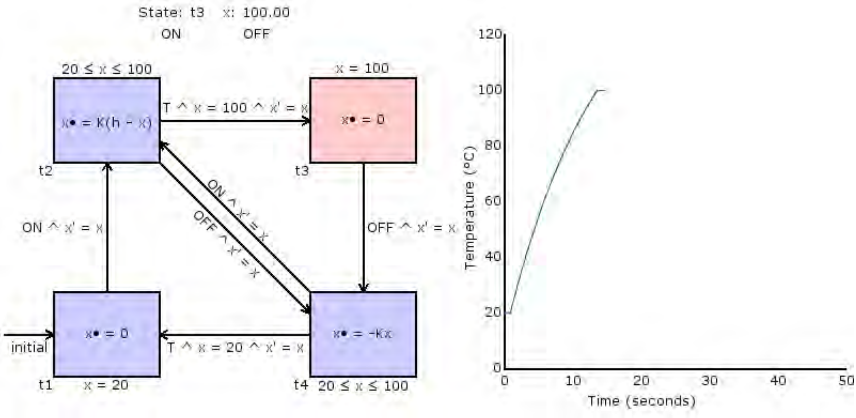
A hybrid automata example



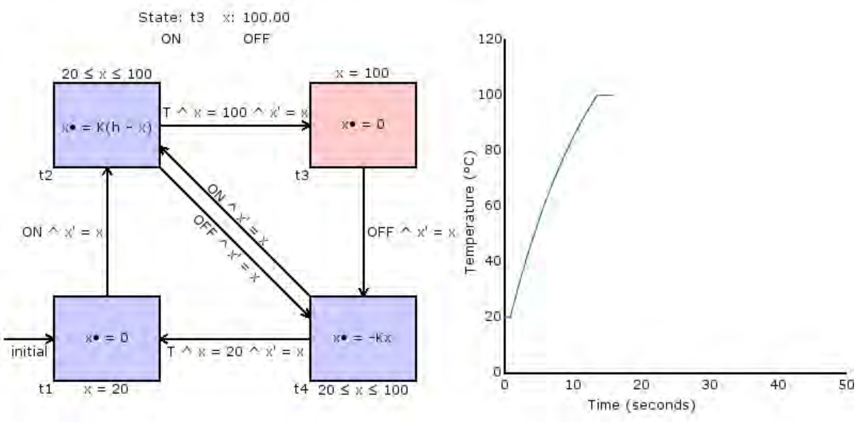


State: t3 x: 100.00
ON
OFF
20 ≤ x ≤ 100
x• = K(h - x)
T ^ x = 100 ^ x' = x
x = 100
x• = 0
t2
t3
ON ^ x' = x
ON ^ x' = x
OFF ^ x' = x
OFF ^ x' = x
initial
x• = 0
T ^ x = 20 ^ x' = x
x• = -Kx
t1
x = 20
t4
20 ≤ x ≤ 100
Temperature (°C)
Time (seconds)
0
10
20
30
40
50
20
40
60
80
100
120

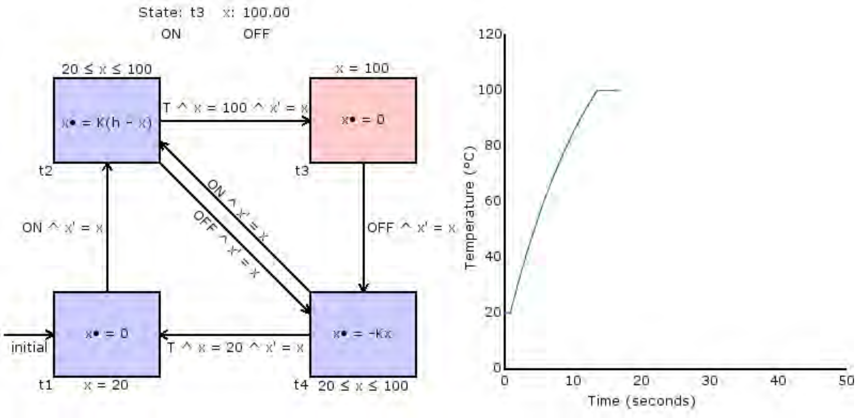
A hybrid automata example



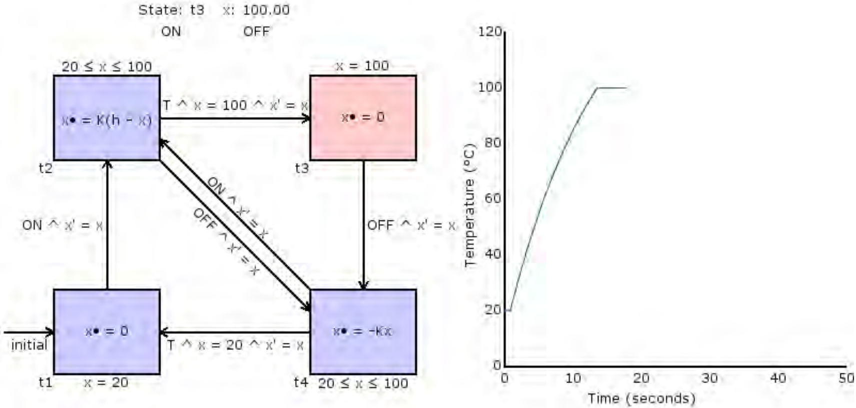
A hybrid automata example



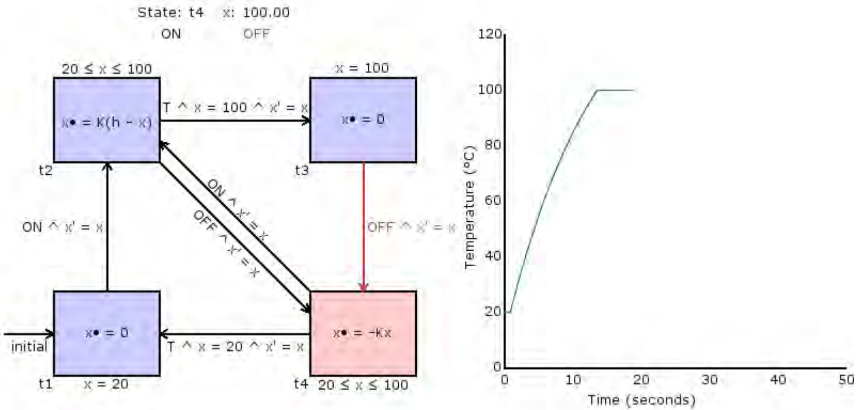
A hybrid automata example



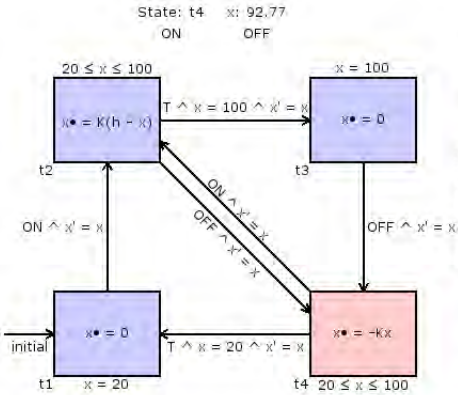
A hybrid automata example

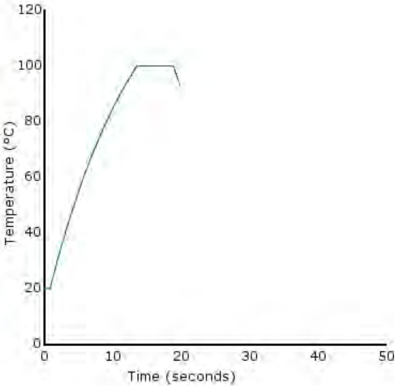


A hybrid automata example

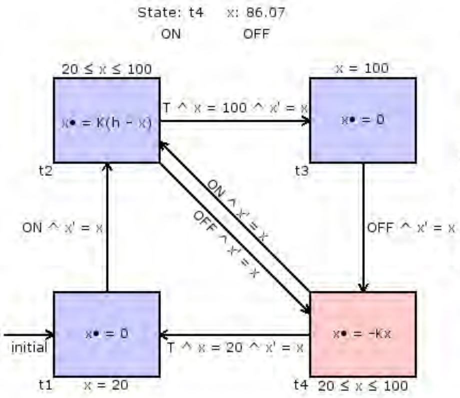


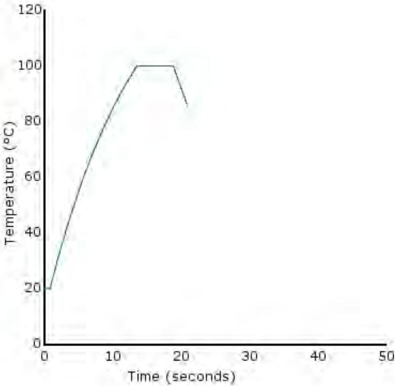
A hybrid automata example



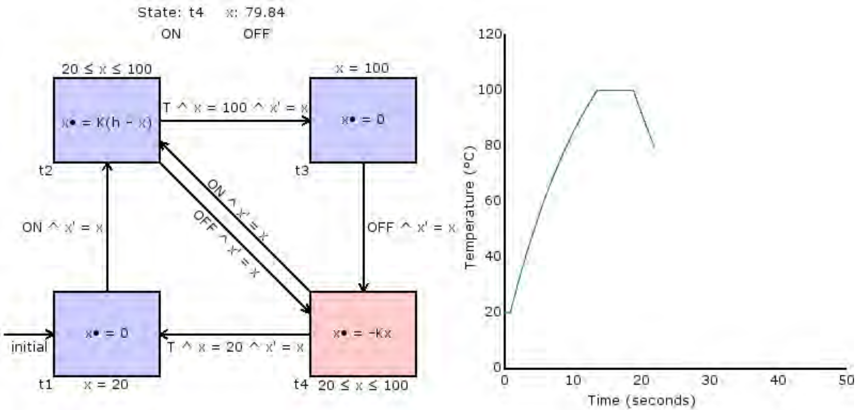


A hybrid automata example

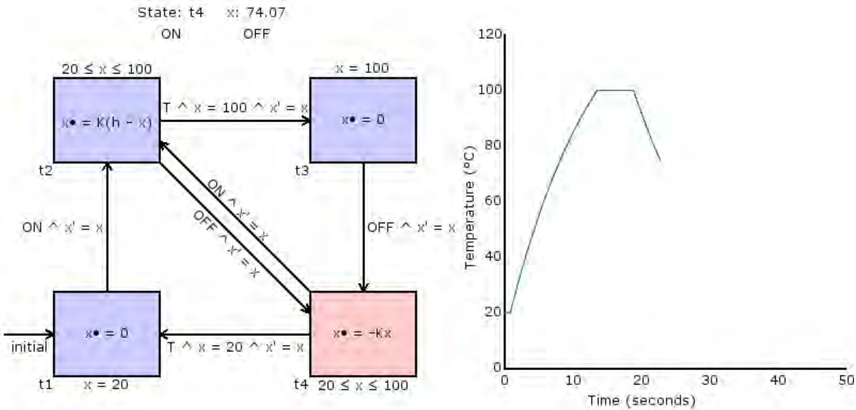




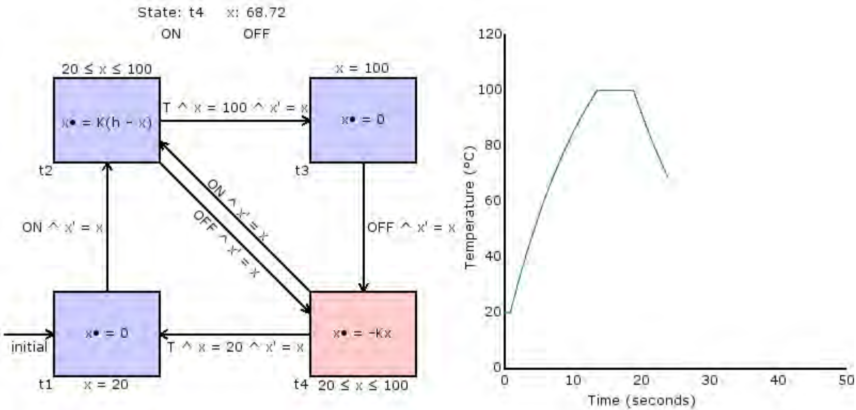
A hybrid automata example



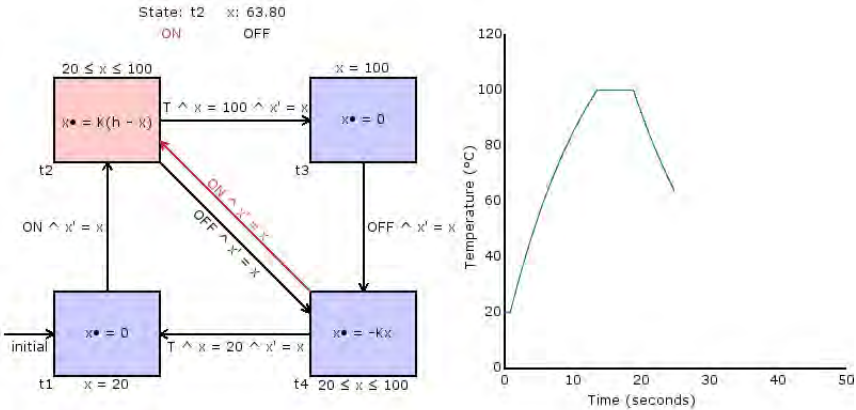
A hybrid automata example



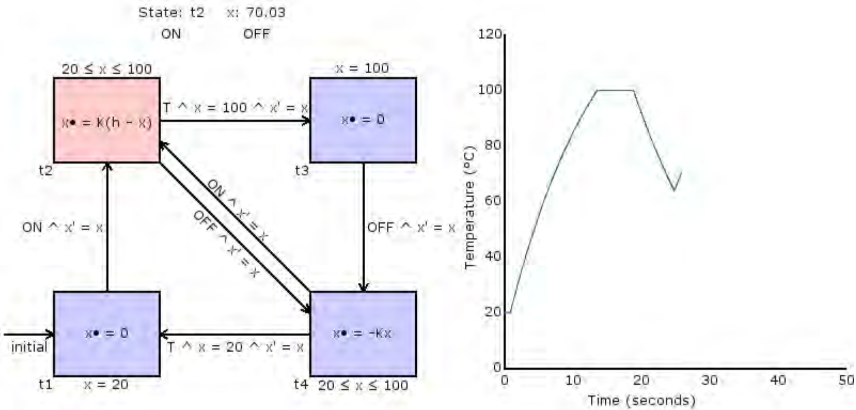
A hybrid automata example



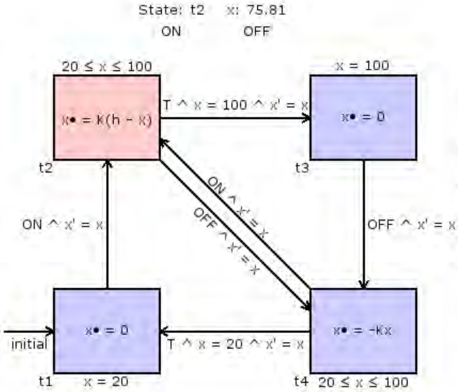
A hybrid automata example

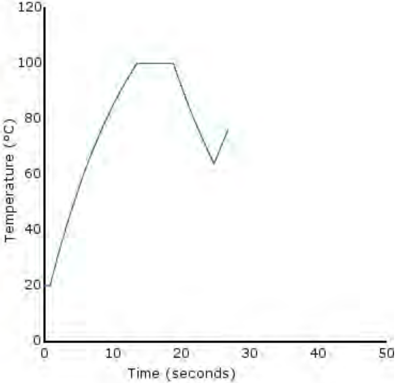


A hybrid automata example

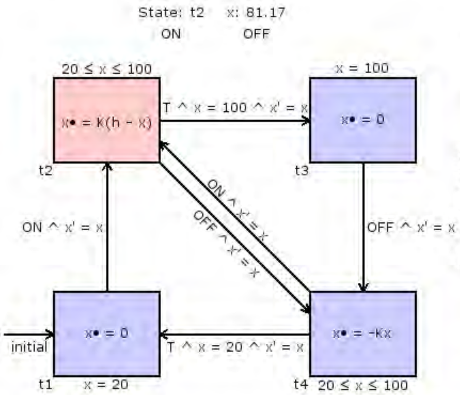


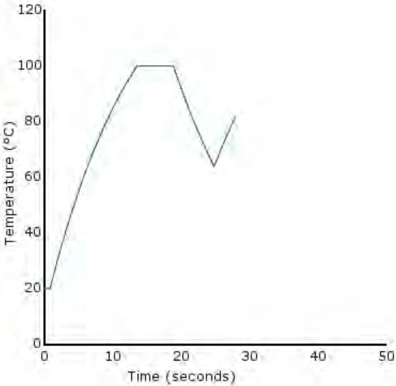
A hybrid automata example



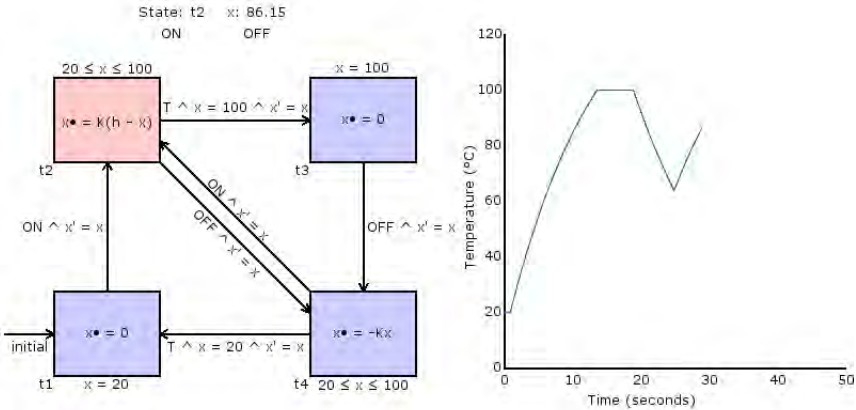


A hybrid automata example

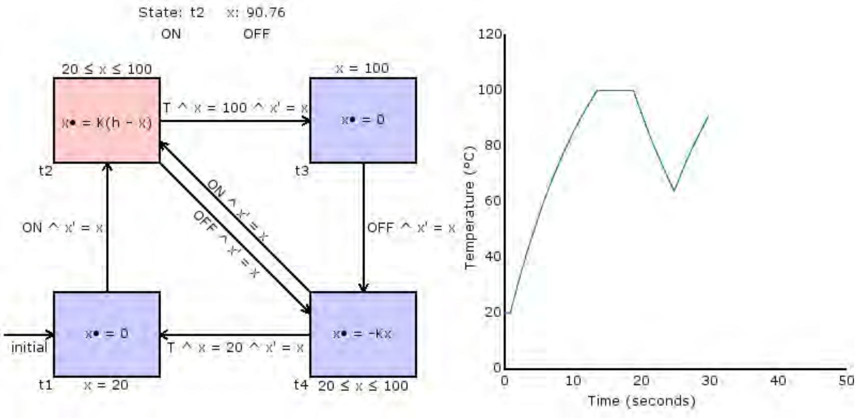




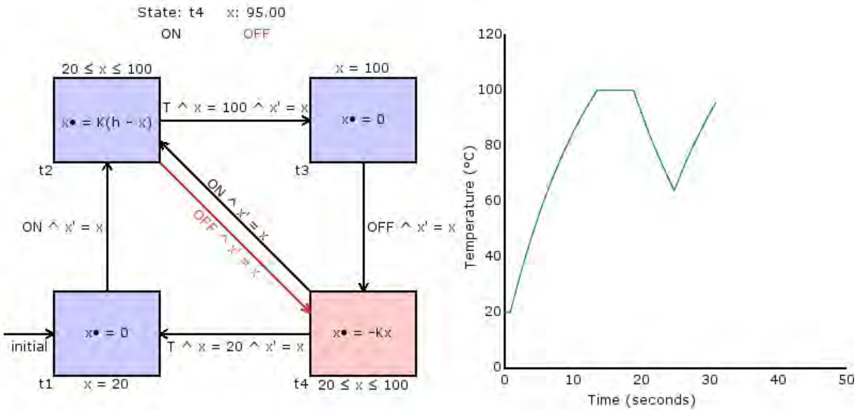
A hybrid automata example



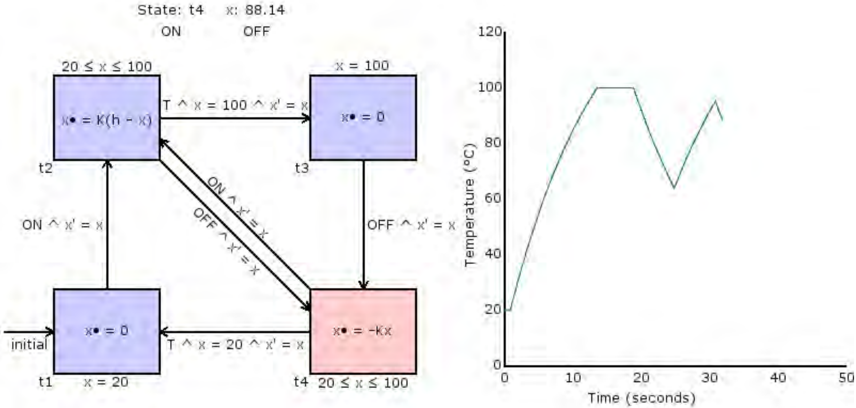
A hybrid automata example



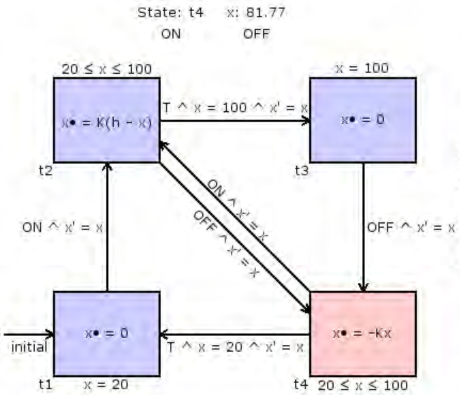
A hybrid automata example

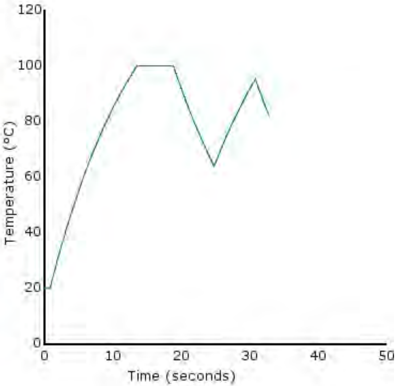


A hybrid automata example

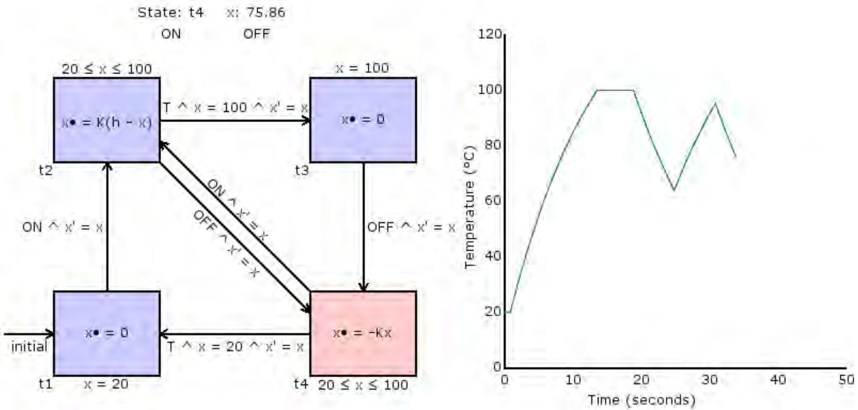


A hybrid automata example

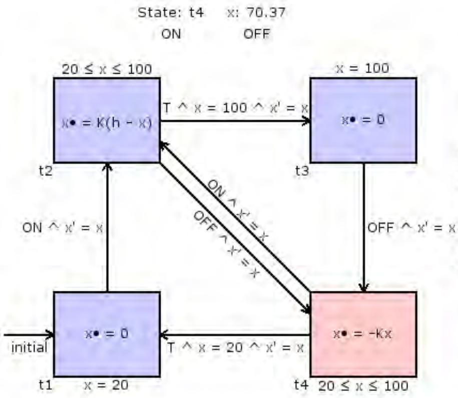


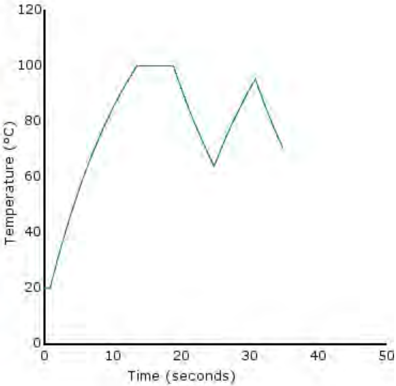


A hybrid automata example

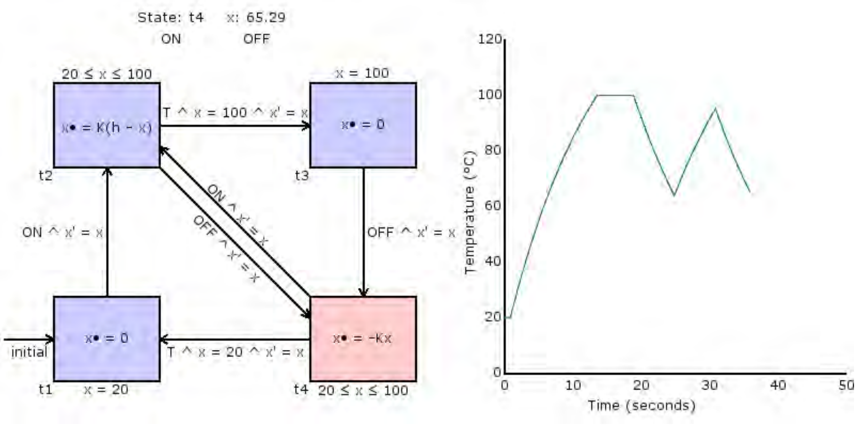


A hybrid automata example

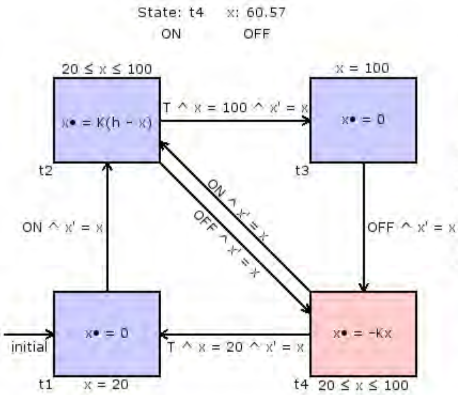


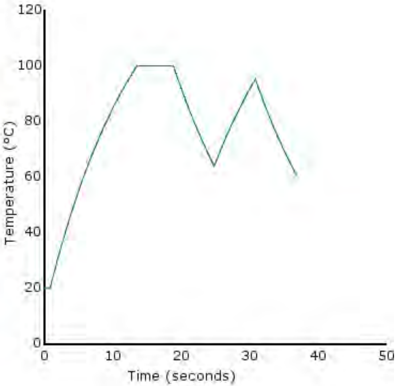


A hybrid automata example

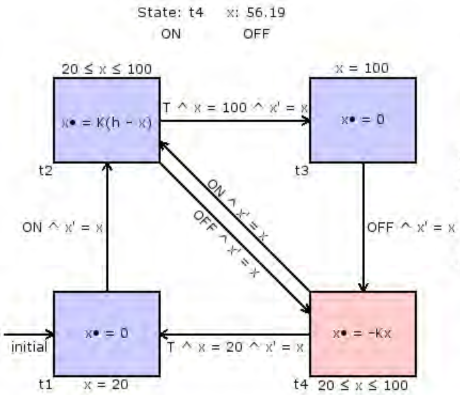


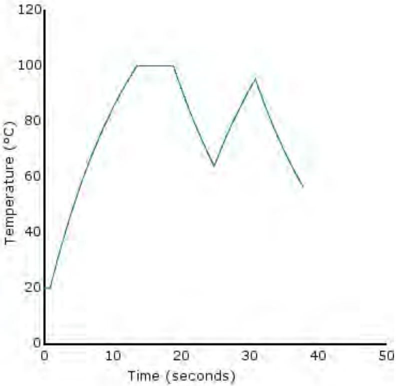
A hybrid automata example



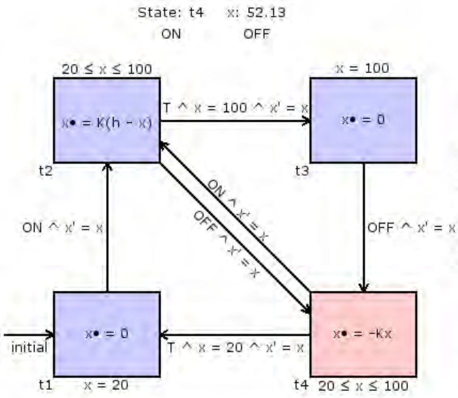


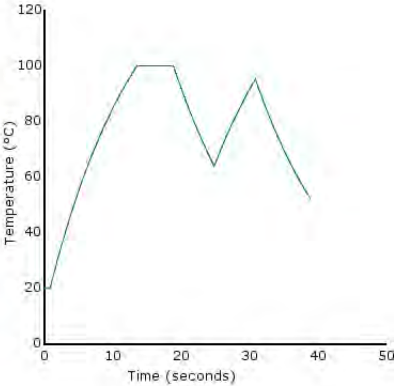
A hybrid automata example



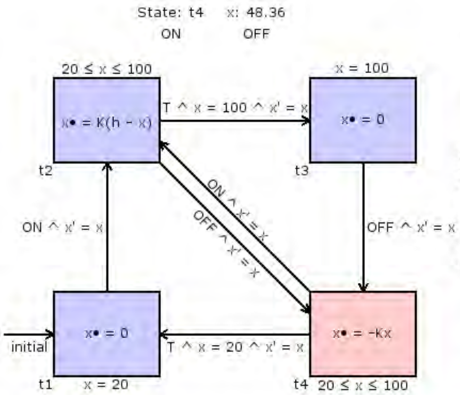


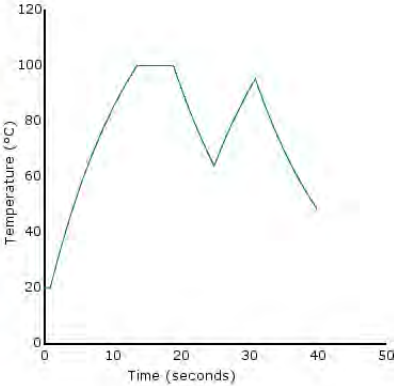
A hybrid automata example



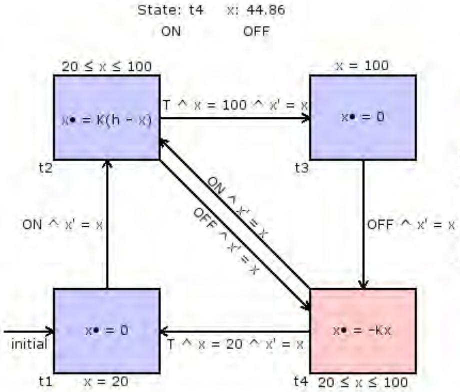


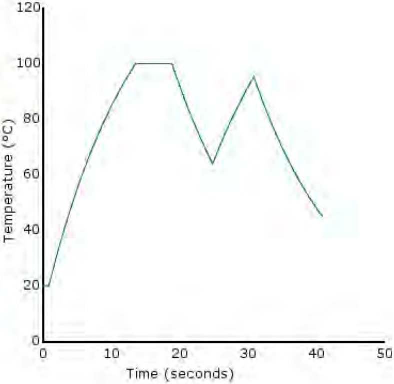
A hybrid automata example



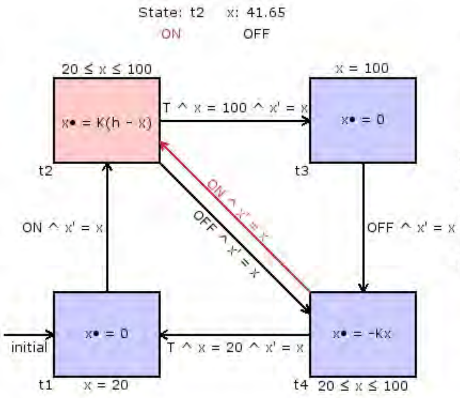


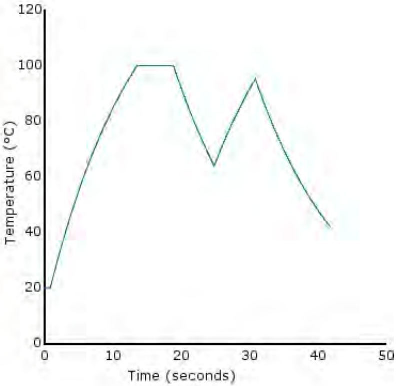
A hybrid automata example



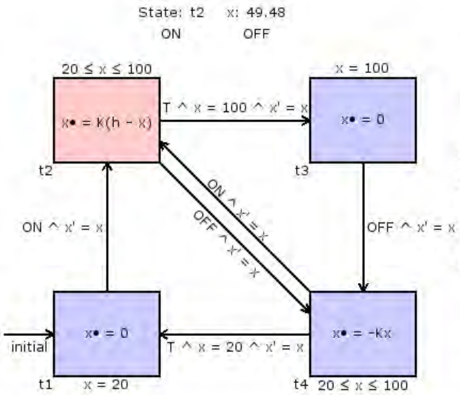


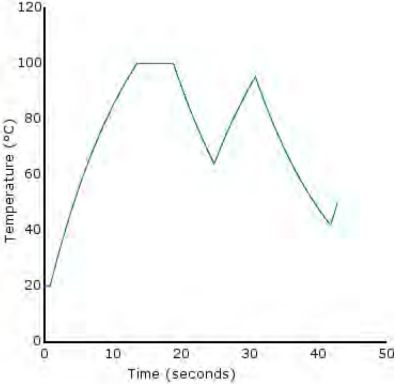
A hybrid automata example



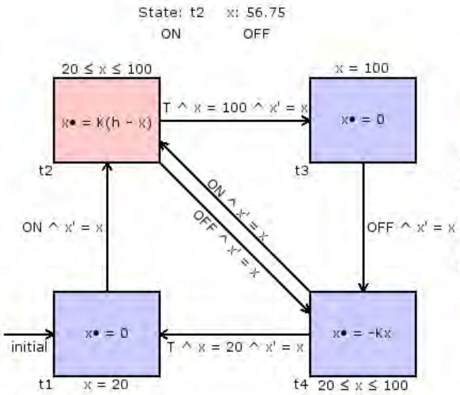


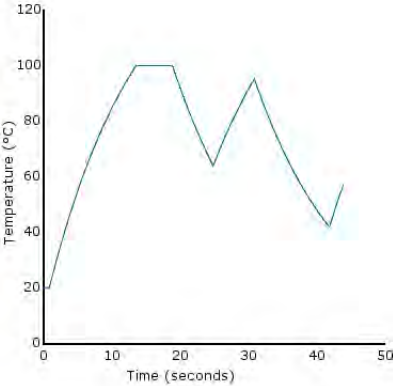
A hybrid automata example



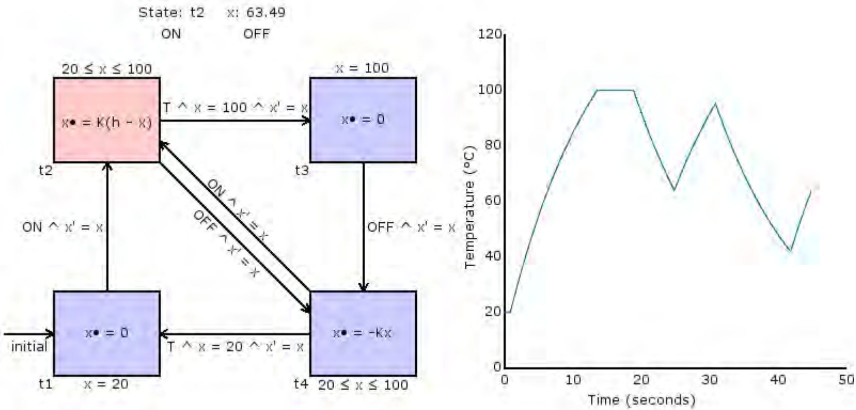


A hybrid automata example

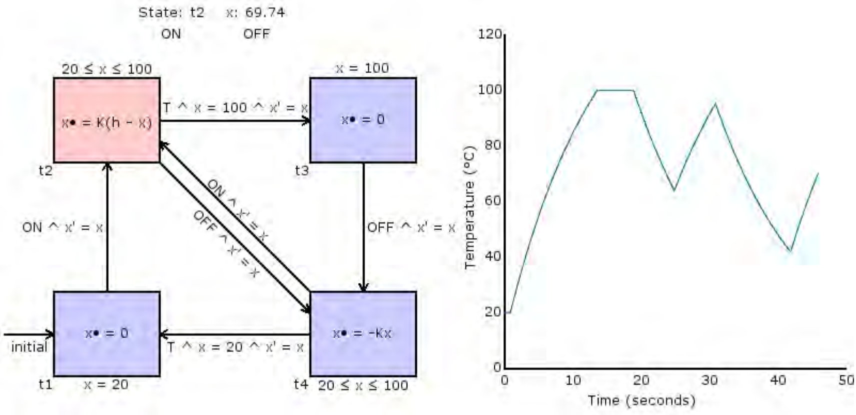




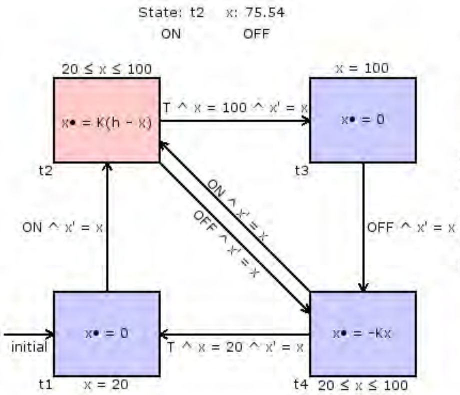
A hybrid automata example

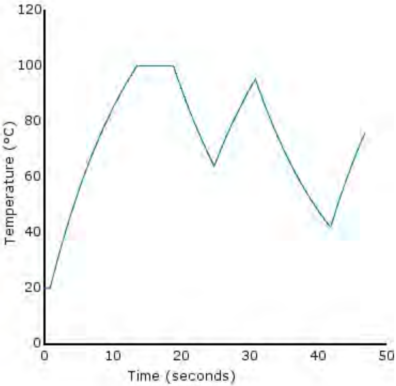


A hybrid automata example

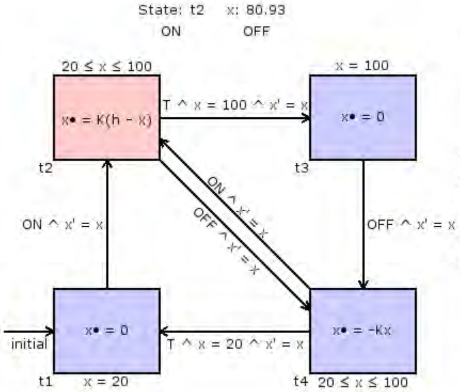


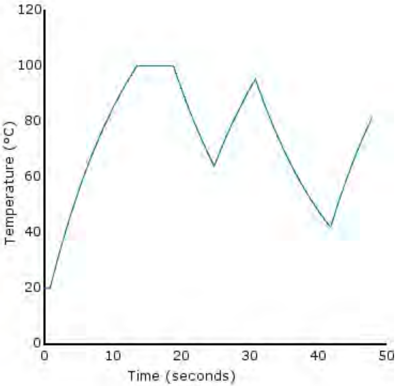
A hybrid automata example



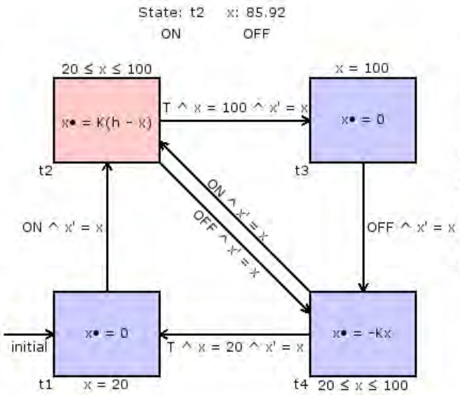


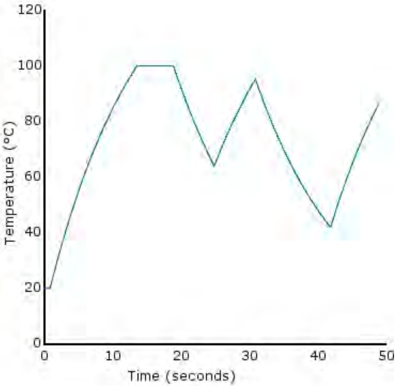
A hybrid automata example



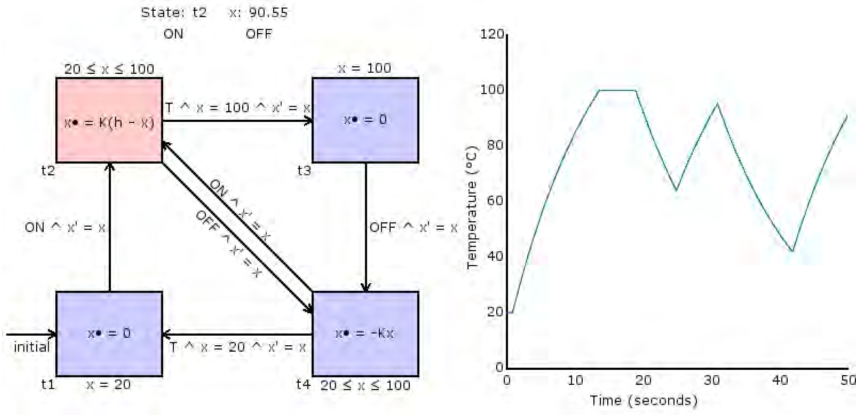


A hybrid automata example





A hybrid automata example



Definition

A hybrid automata H $= < Loc, Edge, \Sigma, Inv, Flow, Jump >$ where

- $Loc = \{l_1, .., l_n\}$ representing n control modes or locations.
- Σ is the input alphabet comprising of event names.
- $Edge \subseteq Loc \times \Sigma \times Loc$ are the set of edges between locations.
- Three sets for the set of continuous variables, their rate of change and their updated values represented as follows: $X = \{x_1, .., x_m\}$ $\dot{X} = \{\dot{x_1}, .., \dot{x_m}\}$ $X' = \{x'_1, .., x'_m\}$.
- $Init(l)$: Is a predicate whose free variables are from X. It specifies the possible valuations of these when the HA starts in l.
- $Inv(l)$: Is a predicate whose free variables are from X and it constrains these when the HA resides in l.
- $Flow(l)$: Is a predicate whose free variables are from $X \cup \dot{X}$ and it specifies the rate of change of these variables when the HA resides in l.
- $Jump(e)$: Is a function that assigns to the edge e a predicate whose free variables are from $X \cup X'$. This predicate specifies when the mode switch using e is possible. It also specifies the updated values of the variables when this mode switch happens.

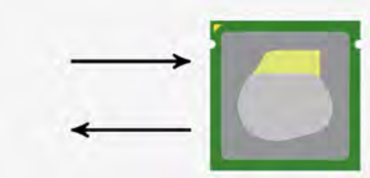
Definition

The semantics of a HA H= $< Loc, Edge, \Sigma, Inv, Flow, Jump >$ is provided using a timed transition system TTA = $< Q, Q_0, \Sigma, \rightarrow>$

- Q is for the form (l, v) where l is a location and $v \in [X \rightarrow R]$ such that v satisfies $Inv(l)$. Q is called the state space of H.
- $Q_0 \subseteq Q$ of the form (l, v) such that v satisfies $Init(l)$.
- $\rightarrow$ is the set of transitions consisting of either:
  - Discrete transitions: For each edge $e = (l, \sigma, l')$, we have $(l, v) \xrightarrow{\sigma} (l', v')$ if $(l, v) \in Q$, $(l', v') \in Q$ and (v, v') satisfy $Jump(e)$. These take zero time.
  - Continuous transitions: When control remains in a location and time progresses. Here the continuous variables evolve according to the ODEs as long as the invariant holds.

Simulation-based validation

- This validation is usually done open-loop.
- System under validation is stimulated using an input trace to observe its response.
- Limitations: Coverage criteria dependent, exhaustive simulation infeasible.



Simulation for validating a pacemaker

What is emulation?

- Operating a **controller** under test in closed-loop with the actual physical process (the **plant or the environment**) [5].
- The design of the controller follows the principles of real-time systems.
- The controller is digital in nature, while the plant usually exhibits continuous dynamics and is uncontrollable.

Emulation for validating a pacemaker
(Actual heart + pacemaker model)

Limitations of emulation

- The plant and the controller may need to be designed in parallel i.e. a rehabilitation robot.
- Model-in-the-loop simulation using Simulink and Stateflow: semantic issues [7, 1] and issues with model fidelity.
- Ptolemy [6] and Zélus [1] are tools with formal semantics. However, these are suitable for the modelling of closed systems using HA models. Also, like SL/SF they interact dynamically with ODE solvers. This is not good for emulation.
- Potemy has incorporated a QSS-based solver [2] to overcome the above. This, however, is unsuitable for open systems.

Problem statement

There is no approach for black-box validation of controllers (say Pacemakers) using real-time plant models. This requires:

- Open models of the plant using a network of hybrid input output automata (HIOA [4]).
- Formal semantics of HIOA models and their compositions.
- Automatic techniques for modular code generation.
- Static timing analysis of the plant for plant-controller timing compatibility i.e. correct timing verification to ensure that the sampling time of the plant and controller match [3]

We propose the new technique of remulation for this.

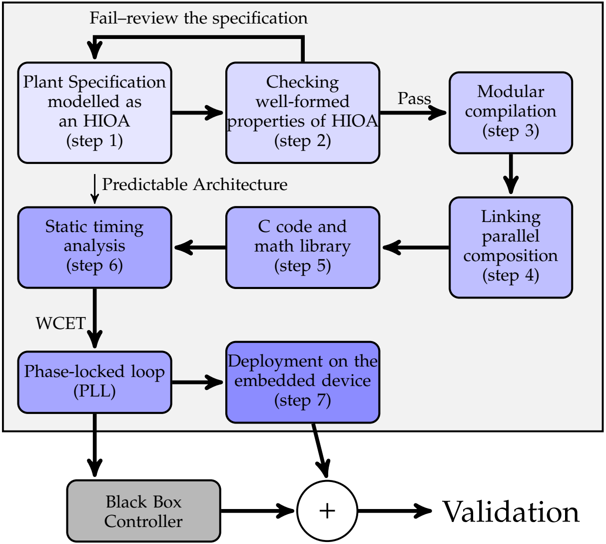
What is remulation?

Remulation stands for reverse emulation using an executable model of the plant, we term a plant-on-a-chip (PoC). During remulation, the plant-controller relationship is reversed.

- We have to synthesize a suitable model of the r-controller (the traditional plant).
- The r-plant (the usual controller) acts as an environment for the r-controller. The r-plant is black-box in nature.

Remulation for validating a pacemaker
(Heart model (real-time) + pacemaker actual/model)

Methodology overview



Overview

1. Introduction
2. Background
3. Motivation / problem statement
4. Methodology
5. Compiling HIOA
   - Compilation overview
   - The first step of the compilation procedure
   - The second step of the compilation procedure
   - Correctly handling the invariant conditions
6. From a cell to the conduction network
   - Models
7. Results

Compilation overview

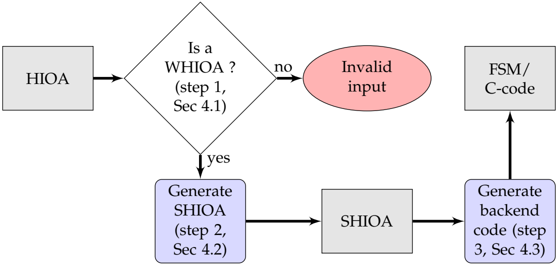


Figure: Overview of the proposed modular code generation approach

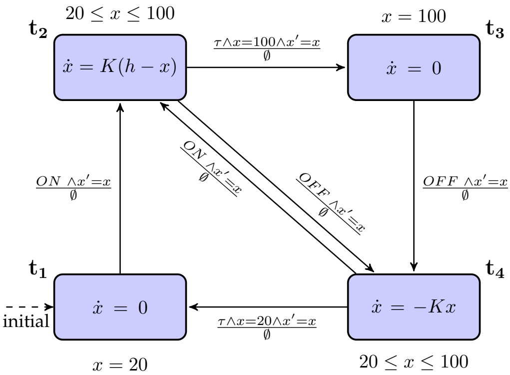
Overview

1. Introduction
2. Background
3. Motivation / problem statement
4. Methodology
5. Compiling HIOA
   - Compilation overview
   - The first step of the compilation procedure
   - The second step of the compilation procedure
   - Correctly handling the invariant conditions
6. From a cell to the conduction network
   - Models
7. Results

Well formed HIOA (WHIOA)

- Discretisation: In HA semantics, when control resides in a location, there are infinite valuations of variables in any given interval of time. This makes code generation difficult. To facilitate code generation, we make evaluations only at discrete intervals. These intervals correspond to the *ticks* of a synchronous program that will be used for code generation.
- Symbolic solution: The ODEs which define flow constraints in any location of the form $\dot{x} = f(x)$ must be of *closed form* nature. This property ensures that such ODEs are symbolically solvable so that the witness functions needed for the generated code are symbolically computable.
- Monotonicity: All witness functions must be monotonic. This property is needed so that the generated code can compute correct valuation of invariants and jump conditions.

Recap: the heating system



Step-1: translating ODEs to witness functions

Symbolic approach using the synchronous abstraction

$$\dot{x} = K(h - x),\ K = 150, h = 0.075 \tag{1}$$
$$x[k] = C_1 \times e^{-0.075 \times \delta \times k} \tag{2}$$

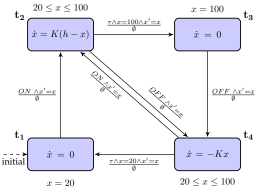
Explanation

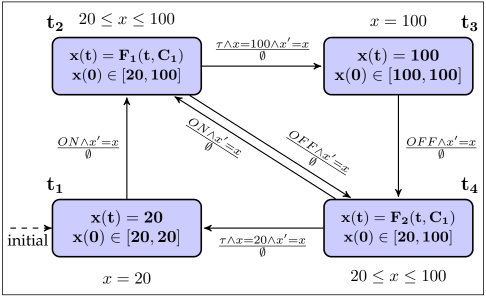
- Equation (1), Ordinary Differential Equation (ODE) captures the evolution of the continuous variable x that represents the temperature in the tank.
- The witness function, for the ODE, is the symbolic (closed form solution) to the ODE, if one exists.
- Equation (2) evolves x *iteratively* while the invariant condition ($10 \leq x \leq 100$) on the location (t_2) holds.
- This iterative evolution of the continuous variables at discrete points in time is akin to transitions on a logical *tick* of a synchronous program.
- We term the Hybrid Input Output Automata (HIOA) obtained after replacing each ODE with its equivalent witness function *Synchronus Hybrid Input Output Automata (SHIOA)*.

Overview

1. Introduction
2. Background
3. Motivation / problem statement
4. Methodology
5. Compiling HIOA
   - Compilation overview
   - The first step of the compilation procedure
   - The second step of the compilation procedure
   - Correctly handling the invariant conditions
6. From a cell to the conduction network
   - Models
7. Results

Compilation step 2: Compiling HIOA to SHIOA

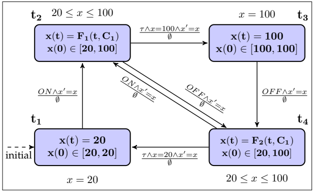


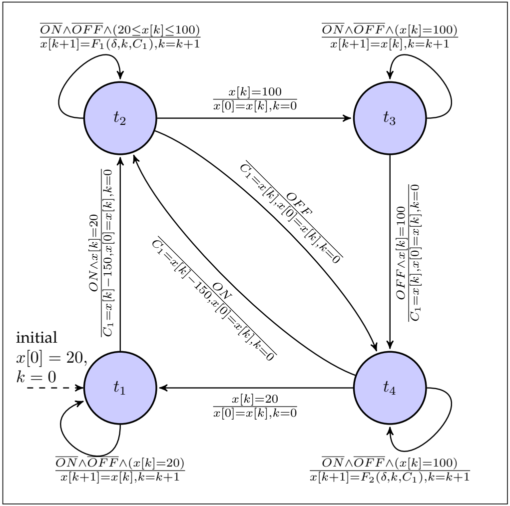


[a]

---

[a] $F1 = C1 \times e^{-0.075 \times t} + 150.0$ and $F2 = C1 \times e^{-0.075 \times t}$





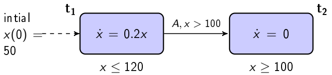
[a]

[a] $F1 = C1 \times e^{-0.075 \times t} + 150.0$
and $F2 = C1 \times e^{-0.075 \times t}$

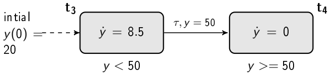
Overview

1. Introduction
2. Background
3. Motivation / problem statement
4. Methodology
5. Compiling HIOA
   - Compilation overview
   - The first step of the compilation procedure
   - The second step of the compilation procedure
   - Correctly handling the invariant conditions
6. From a cell to the conduction network
   - Models
7. Results

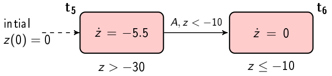
Need for saturation



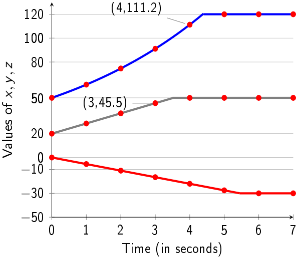
(a) Case 1: an increasing function and it does not need saturation



(b) Case 2: due to equality there is a need for saturation



(c) Case 3: a decreasing function and it does not need saturation



(d) The behaviours of the three example HAs are depicted using solid lines. Our synchronous approximations are depicted using dashed lines. Each tick is one second long.

Figure: The need for saturation depends on the location invariant, the guard in HA, and the step size. Out of the three cases, only Case 2 requires saturation, see Figure 2(d).

Saturation Lemma

Lemma

It is always possible to uniquely determine the saturation value for any continuous variable at time instant k when the state (location) switch from l to l' is to be taken in a SHIOA.

Proof.

The proof of this lemma follows from the following observations.

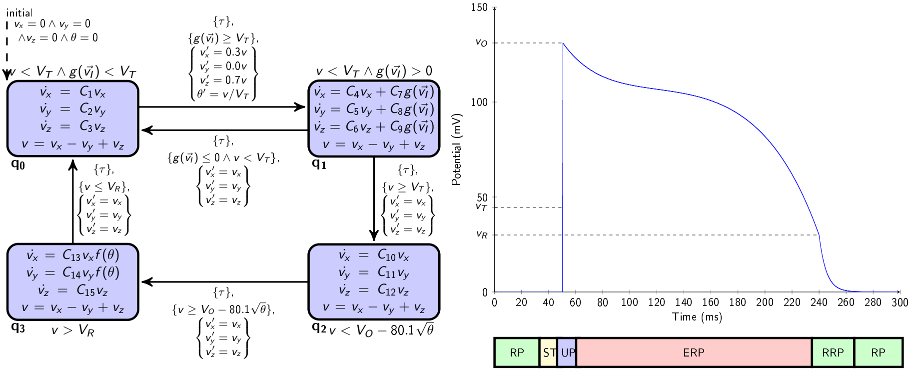
- Observation 1: All witness functions $x(t)$ for any $x \in X$ are monotonic in every location (additional requirement).
- Observation 2: All witness functions $x(t)$ for any $x \in X$ are continuous as they are differentiable in any interval.
- Observation 3: Given the above two observations, the saturation value for any variable x always exists in the time interval $[(k-1) \times \delta, k \times \delta]$ when the location switch happens at instant $k \times \delta$.

□

Overview

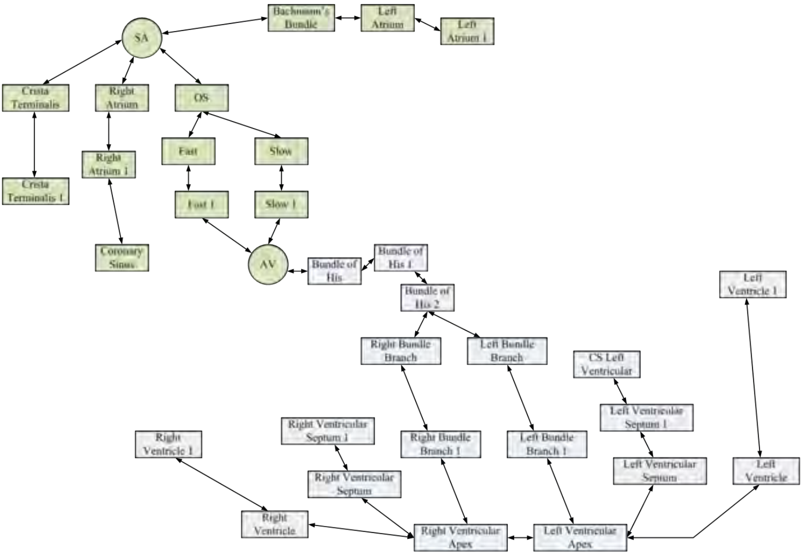
1. Introduction
2. Background
3. Motivation / problem statement
4. Methodology
5. Compiling HIOA
   - Compilation overview
   - The first step of the compilation procedure
   - The second step of the compilation procedure
   - Correctly handling the invariant conditions
6. From a cell to the conduction network
   - Models
7. Results

UoA Model - Cell



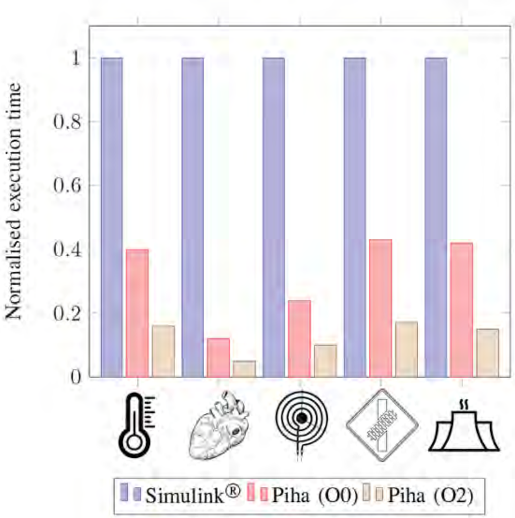
HA based on Ventricular AP.

Conduction Network

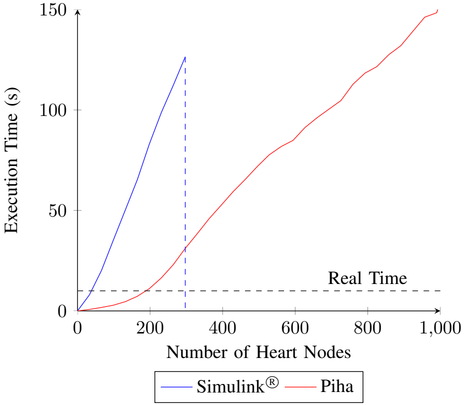


Simulink vs Piha – Execution time

On average 9.8 times faster than Simulink. For the heart conduction system it is two orders of magnitude faster.

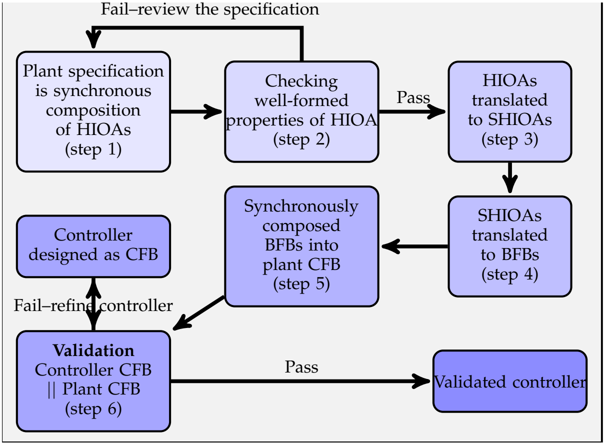


Scalability Relative to Simulink



- 5 times more scalable
- 40 vs 200 cells for real-time emulation

Methodology for PoC design for automation



Conclusions

- We discuss the need for a unified design approach for embedded / automation systems.
- We adopt the well known synchronous approach for designing the controller (using the IEC61499 standard) and the plant (also using the same standard).
- We propose a new technique called reverse emulation (remulation) to design a plant-on-a-chip (PoC).
- A PoC is a model of the plant that offers real-time response similar to the actual plant.
- A key idea is to avoids dynamic interaction with numerical solvers to enable real-time implementations.
- The approach is based on a new class of hybrid automata, we call synchronous hybrid automata.
- We have compared our approach with Simulink and the results are favourable.
- Future work: comparison with Ptolemy and the QSS-based approach,

Key references

- R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Hybrid Systems” , volume 736 of Lecture Notes in Computer Science. Springer-Verlag, 1993.
- E. A. Lee et al. “Modeling and Simulating Cyber-Physical Systems using CyPhySim”, ACM Embedded Software (EMSOFT) conference, 2015.
- N Allen, S. Andalam, P. S. Roop, A. Malik, M. Trew and N. Patel, “Modular code generation for emulating the electrical conduction system of the human heart”, Design Automation and Test in Europe (DATE), Dresden, Germany, 14-18 March 2016.
- A. Malik, P. S. Roop, S. Andalam, E. Yip and M. Trew, “A synchronous rendering of hybrid systems for designing Plant-on-a-Chip (PoC)”, arXiv preprint arXiv:1510.04336 (under review at IEEE Transactions on Software Engineering).