Hybrid System verification in Coq

Hugo Férée
directed by Herman Geuvers

Radbound University, Nijmegen

Abstract. This internship is intended to improve the abstraction method described by Alur in [2], and implemented in Coq in Nimegen [5,7] for proving the safety of hybrid systems.

1 Introduction

Hybrid automata [6] are a good theoretical model for a wide range of real world physical devices. The hybrid system model aims to represent systems measuring continuous physical quantities (*e.g.* temperature, pressure, position or speed) and moving from a state (modes) (among a finite set) to another depending on these values. Then, they can modify some of the environment variables using actuators (*e.g.* heaters, engines) in a way depending on the current discrete state.

There already exists many tools to help proving that the behaviour of the theoretical system (and if it is a good model, of the real system too) is what we intend it to be (in this case, the system is said to be safe). Some of them (*e.g.* Kronos, Uppaal, HyTech) use a symbolic representation of the continuous variables, and then do exact computations, but this requires a lot of assumptions on the system and complex solvers. Other tools, such as CheckMate or d/dt use approximations (with polyhedra or ellipsoids) and optimisation strategies in order to prove the safety of more general systems.

We will here use and try to improve the method described by Alur [2,3] and implemented [5,7] in Coq in Nijmegen using the constructive CoRN library [4] (also developped here in Nijmegen). This method mainly depends on an abstract representation of the hybrid system as a finite graph using a discretization of the continuous space representing the environment variables. Then, the safety of the hybrid system can be proved by looking at the reachability of some nodes of the graph. This is then a proof by computation, and some computations are made over the continuous space. This is why we use the constructive real numbers of the CoRN library. The use of the Coq proof assistant allows to have a certified proof. Then, since Coq computations are not always very efficient,

Ocaml or Haskell code can be extracted to make these computation more quickly.

But this method only underestimates the safety of the hybrid system, and it may fail, for example if this discretization is inappropriate, even if the system is safe. This is why we have tried to make it more robust, by analysing the reason of the failure, and using it to provide a new abstraction which will more likely manage to prove the safety. Of course, this new method won't always work, since even for very simple sets of hybrid systems, the safety is undecidable.

Section 2 describes the hybrid system model. Section 3 gives Alur's abstraction method and gives some important points of Geuvers' Coq implementation, especially how to deal with constructive real numbers in our case. Finally, section 4 describes our improvement of the method.

2 The hybrid system model

2.1 Definition

A hybrid system is a theoretical model, such as finite automata or Turing machines, defined over two kinds of sets:

- a finite set called locations
- an infinite set of points defined by a finite number of continuous variables (typically ranging over $\mathbb{R}$)

A state of the system is then defined by a location and a point.

In the following, we will denote by E the set of points, by L the set of locations, by $S = L \times E$ the set of states and by $\mathbb{R}_{\geq 0}$ the set of positive real numbers.

Like a Turing machine, a hybrid system has a set of initial states, defined by a predicate (called $Init$) and a transition function, but the complexity of the space makes it more comprehensible using several functions and conditions. The transition function is then described by:

- a flow function $\mathcal{F} : S \to \mathbb{R}_{\geq 0} \to E$ (we will denote by $\mathcal{F}_l$ the flow function at location l)
- a predicate $\mathcal{I}$ (the invariant) over S (we will denote by $\mathcal{I}_l$ the invariant predicate at location l)
- a reset function $\mathcal{R} : L \times L \to E \to E$ (we will denote by $\mathcal{R}_{l,l'}$ the reset function from location l to location l')
- a predicate $\mathcal{G}$ (the guard) over $S \times L$ (we will denote by $\mathcal{G}_{l,l'}$ the guard predicate from location l to location l')

where $\mathcal{P}(S)$ is the set of subsets of S.

$\mathcal{R}$ and $\mathcal{G}$ describe the discrete transitions: from the location l and the point p, one can go to the location l' only if the guard $\mathcal{G}((l,p),l')$ is true. Then, the new state is $(l', \mathcal{R}(l,l',p))$.

In addition to the continuous variables, there is also a special time variable ranging over $\mathbb{R}_{\geq 0}$, reset at each discrete transition (*i.e.* it corresponds to the time already spent in the current location). It is needed to define the continuous transitions, described by $\mathcal{F}$ and $\mathcal{I}$:

- the system can stay in the location l only if it verifies the invariant
- if the system entered the location l in point p, then after a duration $d \in \mathbb{R}_{\geq 0}$, it has moved to the point $\mathcal{F}_l(p,d)$

These two kinds of transitions are summed up in figure 1.

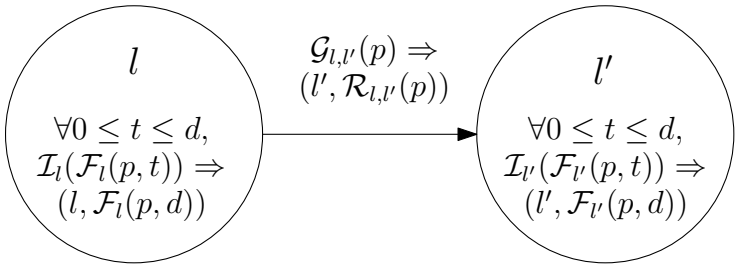

Fig. 1: Schematic representation of the discrete and continuous transitions of a hybrid system.

This is the most general description of a hybrid system, but we can make some restriction, which won't prevent real systems to be modeled.

The intuitive meaning of the flow function it a transformation without memory: the evolution of the system at time t depends on its current state, but not of the previous ones. This property can be expressed by the additivity of the flow function $\mathcal{F}$:

$$\forall l \in L, \forall x \in E, \mathcal{F}_l(x,0) = x$$

$$\forall l \in L, \forall x \in E, \forall t_1, t_2 \in \mathbb{R}_{\geq 0}, \mathcal{F}_l(x, t_1 + t_2) = \mathcal{F}_l(\mathcal{F}_l(x,t_1), t_2)$$

These are examples of flow functions:

Example 1. Flow functions

– $x, t \to x$ (constant flow)
– $x, t \to x + \alpha t$, where $\alpha \in \mathbb{R}$ (linear flow)
– $x, t \to xe^{\alpha t}$ (exponential flow)

2.2 The thermostat example

The following example (described by figure 2) has been studied in [5,7]. The thermostat is a typical example of hybrid system if we see it as a system measuring some environment variables (here the temperature T and a timer c) and acting on them (here heating or cooling the environment).

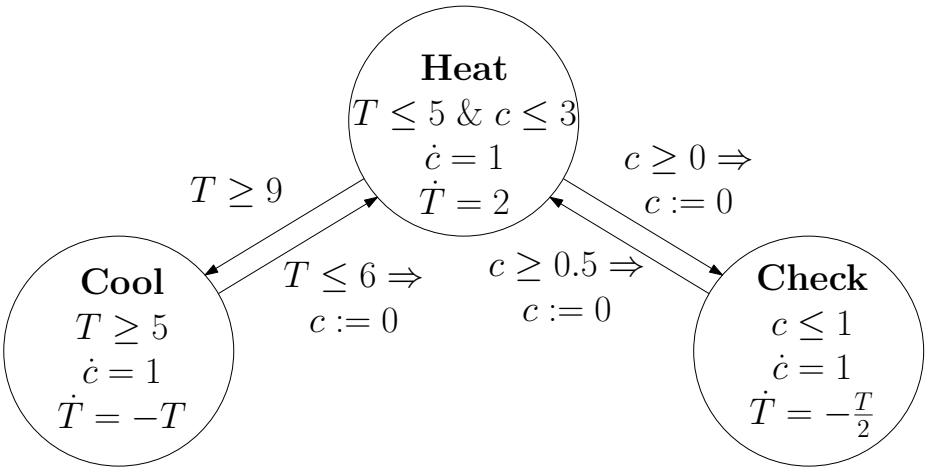

Fig. 2: The thermostat example

Notice that on figure 2, the flow function is described by differential equations ($\dot{T}$ represents the derivative of T). For example, the equation $\dot{c} = 1$ defines the linear flow with $\alpha = 1$, and $\dot{T} = -T$ defines the exponential flow with $\alpha = -1$. This is the case in most concrete examples and one could restrict to linear differential equations without real loss of generality.

2.3 safety

Once the hybrid system is defined, the main goal is to prove its safety, *i.e.* that it evolves as we intend it to. This safety is defined by a predicate $Safe$, or equivalently as its complement, $Unsafe$. For example,

we may want that our thermostat keeps the temperature above $5°C$ (*i.e.* $Unsafe(l,(c,T)) = T \leq 5$) if the initial temperature is between 5 and 10 degrees, the timer is 0 and the initial location is **Heat** (*i.e.* $Init(l,(c,T)) := (l = \textbf{Heat}) \wedge (c = 0) \wedge (5 \leq T \leq 10)$).

Proving the safety of a given hybrid system might be quite difficult, even though it is quite simple to prove the safety of the thermostat example by hand. But when the number of locations increases, this proof can become tedious, and we might want a real guarantee of the correctness in concrete examples. It has been proven [1] that the safety is undecidable even for very simple classes of hybrid systems (*e.g.* linear hybrid systems, with linear reset and flow functions and linear inequalities for the guard, invariant, initial predicate and unsafe predicate). So we can not hope to have a complete solver.

Notice that the behaviour of a hybrid system is non-deterministic: for example, in the thermostat system, one can go from location **Heat** to location **Cool** whenever the temperature is between 5 and 6 and that there are, in the general case, uncountably many possible traces from an initial state. This is why a solution to prove its safety is to discretize the space of the hybrid system, which is the main idea of the method we will describe in the next section.

3 The abstraction method

3.1 Description of the method

Alur [2] has described a method to prove the safety of hybrid systems. It consists in reducing the problem of the reachability of unsafe states in the hybrid system into the reachability of some nodes of a finite graph called abstract hybrid system, build from the inital concrete hybrid system using this method:

- divide the point space into a finite number of (simple) subsets called abstract regions
- build a graph whose nodes are pairs of location and abstract regions (then called abstract states) and its edges overestimate the concrete transitions
- define a set of abstract states overestimating the inital states and a set of abstract states overestimating the unsafe states
- check that the unsafe abstract states are not reachable from the abstract initial states in this graph

Overestimating the transitions means that if there is a (discrete or continuous) transition from a concrete state A to a concrete state B, then there exists an edge in the abstract system from an abstract state containing A to and abstract state containing B.

Then, if a point is reachable in the concrete system, then it is included in a reachable abstract region. Thus it is straightforward that if the unsafe abstract states are not reachable, then the concrete system is safe.

Though, it might be difficult to find a good division of the space and precise enough over-estimators for the transitions. Nevertheless, if the guard, the invariant and the reset function are linear, a good heuristic [3] is to divide the continuous space into a regular grid described by the corresponding linear coefficients.

Alur's sketch of implementation uses floating point numbers to represent real numbers, and we might want to use a certified program to prove the safety of a hybrid system. This method has been implemented in Coq [5,7] using the the computational power of the constructive real numbers of the CoRN constructive mathematics Coq library [4]. We will here discuss some of the difficulties encountered while implementing this method.

3.2 Constructive reals, approximations and the double negation

In CoRN, a representation of a real number x is isomorphic to a sequence of arbitrary rational approximations of x (and this is equivalent to most definitions of computable real numbers). But we cannot decide whether $a < b$ or $b \leq a$ for $a, b \in \mathbb{R}$. Then, we cannot decide whether a point of the continuous set of points is in a given region (typically a rectangle) or not. Nevertheless, we can say if a given approximation of the point is or not in the rectangle. But then, different representations of the same real number might be assigned to different rectangles, and points outside the rectangle but close enough can be said to be in it. The solution is to use the double negation monad:

$$\forall x, x \to \neg\neg x \ (return_{DN})$$

$$\forall xy, \neg\neg x \to (x \to \neg\neg y) \to \neg\neg y \ (bind_{DN})$$

Indeed, $\neg\neg(a < b \vee b \leq a)$ is provable, and we can keep doing our proof under this monad (using $bind_{DN}$). We only need the invariant to be stable (*i.e.* provably equivalent to its double negation). Typically, non strict inequalities, or properties with a head negation are stable, but strict

inequalities are not. Then, we can deal with invariants defined by conjunctions of large inequalities, which a reasonable assumption. Finally, we can get out of this double negation since our goal is to prove that unsafe states are not reachable, which is a negation, and is then stable.

Sometimes, this is not possible and we need to use overestimations parameterized with a precision. An overestimation of a property P is an element of the (Coq) set $\{b : bool | b = false \rightarrow \neg P\}$. For example, we can write a function, returning true at least when the real number represented by the input is in a given square, and outputs false at least when it is at distance at least ε of the square, where $\varepsilon > 0$ is the precision. This is an overestimation of the property "being in this square", and we can make it as accurate as we need, depending on the parameter ε. This parameter may be critical for the success of the method.

This is useful in particular to inverse the flow function. Indeed, to overestimate continuous transitions, one need to inverse it to know if, for two squares and at a location l, does it exist a duration $d \in \mathbb{R}_{\geq 0}$ such that the image of one square by $\mathcal{F}_l(., d)$ intersects the second one. If yes, then there need to be an abstract transition between these abastract states.

To do so, we assume that the flow function is separable:

Definition 1 *Separable function* $F : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0} \Rightarrow \mathbb{R}^n$ *is separable if:* $\forall x_1, \dots x_n, d, \mathcal{F}((x_1, \dots, x_n), d) = (\mathcal{F}_1(x_1, d), \dots \mathcal{F}_n(x_n, d))$

This means that the flow function can be decomposed into one-dimensional flow functions over each axis. This is restrictive hypothesis, used to simplify the implementation. We believe that it could be removed in the Coq implementation, but it would be a lot of work.

Then, we assume that each projection of the flow function is range-invertible:

Definition 2 *Range inverse* $\mathcal{F}^{-1} : OpenRange \rightarrow OpenRange \rightarrow OpenRange$ *(where OpenRange describes the set of intervals of* $\mathbb{R} \cup -\infty, +\infty$ *with bounds included when they exist) is a range inverse function for the function* $\mathcal{F}$ *if:*

$$\forall (I, J : OpenRange), \forall p \in I, \forall d, \mathcal{F}(p, d) \in J \Rightarrow d \in \mathcal{F}^{-1}(I, J)$$

Then, if we only consider separable flow functions (as defined below), we only have to check if these three intervals overlap to know if there must be a transition between the squares $I_1 \times I_2$ and $J_1 \times J_2$ (see figure 3 for an illustration):

$$\mathcal{F}_x^{-1}(I_1, J_1) \cap \mathcal{F}_y^{-1}(I_2, J_2) \cap \mathbb{R}_{\geq 0} \neq \emptyset$$

, where $\mathcal{F}_x$ and F_y are two one-dimensional flow functions such that $\mathcal{F}((x,y),d) = (\mathcal{F}_x(x,d), \mathcal{F}_y(y,d))$. This can also be overestimated with a fixed precision $\varepsilon > 0$.

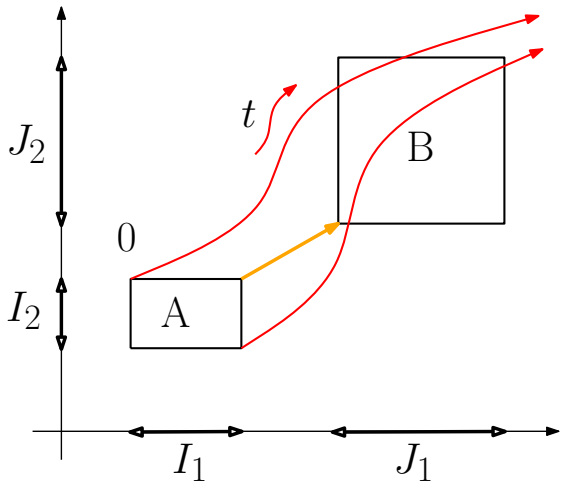

Fig. 3: Two abstract squares(A and B) the image of A by $\mathcal{F}$ intersects B for some durations included in $\mathcal{F}^{-1}(A,B)$.

3.3 Other implementation details

In order to make it easier to use, this implementation includes a small library of flow functions, in particular those of example 1.

Once the abstract system is defined and the abstract transitions are computed, we can run a proved Dijkstra's (breadth-first search) reachability algorithm on the abstract graph, which will tell us whether the abstract system is safe or not. If it is, then the concrete system is also safe. If it is not, then one cannot say anything. Next section will describe what we can do in the last case (and if we are indeed convinced of the safety of the system).

4 Refining the method

4.1 Refinement idea

If the previous method fails, and it can easily happen even if the system is simple, then you can use more accurate overestimations (*i.e.* basically

use a smaller precision parameter ε), but this is, most of the time not sufficient. Indeed, the key issue is to find the right abstraction (*i.e.* grid).

The first idea is to rewrite the reachability algorithm in order to provide a reachability trace (*i.e.* a path from an initial state to an unsafe one) for each reachable unsafe state. If the concrete system is safe, then this means that somewhere on this trace something went wrong, *i.e.* there is an abstract state on this path containing (or at a distance smaller than the precision of the over-approximations) both reachable (concrete) states and states from where unsafe states are reachable (which we will call pre-unsafe in the following).

Then, the solution would be to divide each square on this path. But in the worst case (and in most cases), this will mean that we will use a grid with about 2^n times more squares (where n is the dimension of the continuous space) if we need to divide the grid everywhere.

This would be a lot of waste of space and computation time and this is why we use a more flexible abstraction, in order to divide the squares only where we think it is necessary. In dimension 2, which will be the case in the following for the sake of simplicity, we will use quadtrees (it would be octrees in dimension 3 for example). In our point of view, a quadtree is a tree of arity four, each node being labeled with a point and implicitly dividing the plane into four sub-planes.

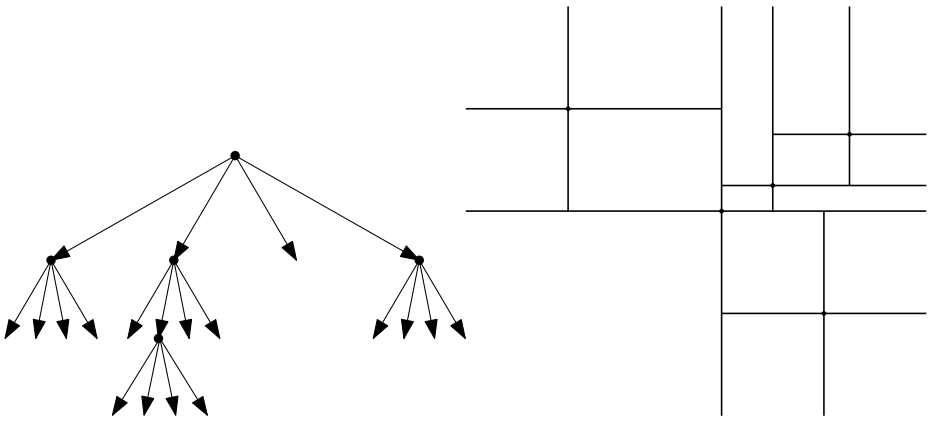

Fig. 4: Example of quadtree (left) and the associated division of the plane (right).

Figure 4 gives a representation of a quadtree (as a tree and the implicit division of $\mathbb{R}^2$). Notice that replacing a leaf of the tree with a node divides the corresponding square.

Figure 5 shows an abstraction making some unsafe states reachable (since the set of reachable concrete states and the set of pre-unsafe overlap a common abstract region). Then, we can refine the grid as in figure 6, but the same problem occurs, and another refinement would be necessary, making the number of abstract states increase too much, and so the computation time.

On the contrary, figure 7 shows a possible refinement of the grid seen as a quadtree. The refinement is made only where it seems necessary. We still need several refinement steps, but the number of abstract states keeps reasonably low.

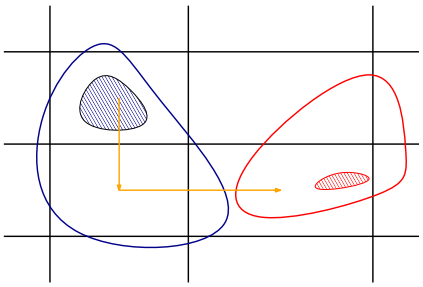

Fig. 5: The grid of abstract squares (in black), the initial and reachable states (in blue) and the final and pre-unsafe states (in red). The initial and unsafe sets are dashed, respectively in blue and red. The bottom-right square intersects both reachable and pre-unsafe sets, making the abstract system unsafe.

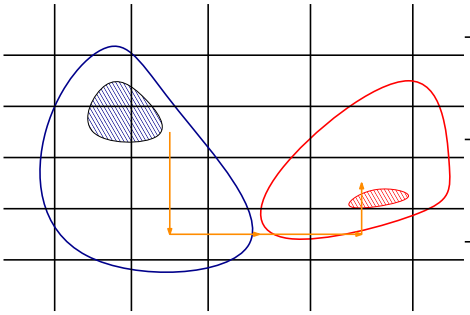

Fig. 6: The refined grid.

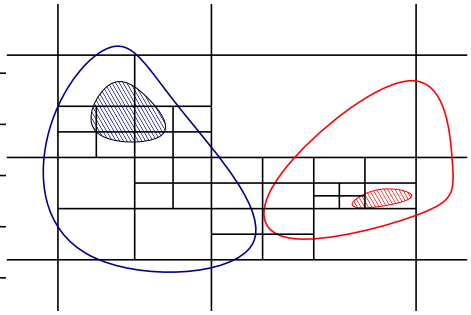

Fig. 7: The refined quadtree (several iterations).

The refinement with the grid abstraction is about brute force, while the refinement of the quadtree abstraction is more clever, but might need a lot more refinement steps. Then, it is not obvious that this abstraction will be more efficient.

4.2 Further work

I have implemented this new abstraction in Coq (including the new version of the proved Dijkstra's reachability algorithm), trying to make it equivalent (in terms of provability and efficiency) to the previous grid abstraction. There still need to do some example, especially the thermostat example, in order to compare in practice the relative efficiency of the two methods.

We also need to try it on examples where the method initially fails, to do benchmarks on the whole refinement process, comparing the refinement of the grid (which might take less refinement steps) and the refinement of the quadtree which might take less computation time at each refinement), assuming that the refinement iteration process terminates.

Indeed, if we take an example where the reachable states cover the half plane defined by $x \leq 0$, and the unsafe states are defined by $0 < x$, then there is no such abstraction allowing to prove the safety of the abstract system. But this corresponds to an unstable system, since in real systems, measurements cannot be made exactly. This is why the use of computational real numbers is not restrictive since it models well these concrete situations.

Then, we may want to have sufficient conditions for the termination of this method. In particular, we conjecture that for an abstract system with bounded reachable and pre-unsafe sets, with non zero distance (which might be the case if we only consider large inequalities in the predicates) and for a sufficiently precise precision parameter (our previous ε), the refinement process terminates. This is intuitively justified by the fact that each time a path from an initial state to an unsafe state is refined, then somewhere on the path a state containing reachable and pre-unsafe states is refined, and these two sets will finally be covered by two disjoint sets of abstract states. Figure 8 shows such a system. The separation distance (here ε), and a bound on the size of the reachable and pre-unsafe states could be used to find an upper bound on the number of refinement steps and more generally on the complexity of the process. Indeed, the more these sets are far fromeach other, the easier they can be separated by disoint union of the squares of a quadtree.

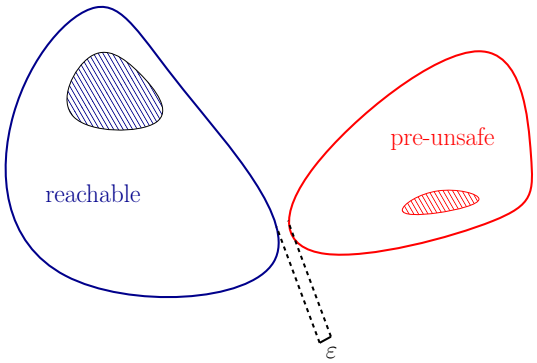

Fig. 8: A provably safe system (according to our claim):reachable and pre-unsafe sets are separated by ε).

5 Conclusion

The whole work of automation of the process is still not sufficient to use it for real cases. The aim of this work is more intended to show that we can use constructive real numbers to do proofs by computation in Coq.

More than implementation improvements and benchmarks, this works needs a few results such as our previous claim. But these conditions require to know the set of reachable states and pre-unsafe states, which is almost equivalent to proving the safety by hand.

Then, it would be more interesting to have implicit conditions for the termination of the process, and having an implicit bound on the number of refinement steps could also allow to prove the unsafety of the system in some cases if the abstract system is still unsafe after this number of steps.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995. Hybrid Systems.
2. R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. *Hybrid Systems: Computation and Control*, pages 758–819, 2002.
3. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 5(1):152–199, 2006.
4. L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

5. H. Geuvers, A. Koprowski, D. Synek, and E. van der Weegen. Automated Machine-Checked Hybrid System Safety Proofs.
6. T. Henzingerz. The Theory of Hybrid Automata y. Technical report, Citeseer.
7. E. van der Weegen. Automated Machine-Checked Hybrid System Safety Proofs.