Automated Machine-Checked Hybrid System Safety Proofs*

Herman Geuvers[1,2], Adam Koprowski[3], Dan Synek[1], Eelis van der Weegen[1]

[1] Radboud University Nijmegen
[2] Technical University Eindhoven
[3] MLState, Paris

Abstract. We have developed a hybrid system safety prover, implemented in Coq using the abstraction method introduced by [2]. The development includes: a formalisation of the structure of hybrid systems; a framework for the construction of an abstract system (consisting of decidable "over-estimators" of abstract transitions and initiality) faithfully representing a concrete hybrid system; a translation of abstract systems to graphs, enabling the decision of abstract state reachability using a certified graph reachability algorithm; a proof of the safety of an example hybrid system generated using this tool stack. To produce fully certified safety proofs without relying on floating point computations, the development critically relies on the computable real number implementation of the CoRN library of constructive mathematics formalised in Coq. The development also features a nice interplay between constructive and classical logic via the double negation monad.

1 Introduction

In [2], Alur et al. present an automated method for hybrid system safety verification in which one derives from the hybrid system of interest an *abstract* hybrid system, which is essentially a finite automaton whose traces are sufficiently representative of traces in the original system that unreachability in the abstract system (which can be decided using a standard graph algorithm) implies unreachability in the concrete system (which, being governed by continuous behaviours, cannot be decided so readily). Thus, the abstraction method brings the safety verification problem from a continuous and infinite domain into a discrete and finite domain, where it can be dealt with using standard graph algorithms.

The prototype implementation described in [2] was developed in a conventional programming language, only has an informal correctness argument, and uses ordinary floating point numbers to approximate the real numbers that are used in said argument. These factors limit the confidence one can justifiably have in safety judgements computed by this implementation, because (1) it is easy for bugs to creep into uncertified programs; (2) it is easy to make mistakes in

* This research was supported by the BRICKS/FOCUS project 642.000.501, Advancing the Real use of Proof Assistants

informal correctness arguments; and (3) floating point computations are subject to rounding errors and representation artifacts.

Our goal is to increase this degree of confidence by developing a *certified* reimplementation of the abstraction technique in Coq, a proof assistant based on a rich type theory that also functions as a (purely functional) programming language. The Coq system lets us develop the algorithms and their formal correctness proofs in tandem in a unified environment, addressing (1) and (2) above.

To address (3), we replace the floating point numbers with exact computable reals, using the certified exact real arithmetic library developed by O'Connor [14] for CoRN, our Coq repository of formalised constructive mathematics [7]. This change is much more than a simple change of representation, however; because computable reals only permit observation of arbitrarily close approximations, certain key operations on them (namely naive comparisons) are not decidable. The consequences of this manifest themselves in our development in several ways, which we discuss in some detail. Hence, our development also serves to showcase O'Connor's certified exact real arithmetic library applied to a concrete and practical problem.

On a separate note, we argue that the use of computable reals is not just a pragmatic choice necessitated by the need to compute, but is actually fundamental considering their role in hybrid systems, where they represent physical quantities acted upon by a device with sensors and actuators. In the real world, measurements are approximate.

The end result of our work is a framework with which one can specify (inside Coq) a concrete hybrid system, set some abstraction parameters, derive an abstract system, and use it to compute (either inside Coq itself or via extraction to OCaml) a safety proof for the concrete system.

2 Hybrid Systems and the Abstraction method

A hybrid system is a model of how a software system (running on a device with sensors and actuators), described as a finite set of *locations* with (discrete) transitions between them, acts on and responds to a set of continuous variables (called the *continuous state space*), typically representing physical properties of some environment (such as temperature and pressure).

There are many varieties of hybrid systems [9,11]. We follow [2] and to illustrate the definition and the abstraction method, we use the example of a system describing a thermostat (this is the same example as in [2]), shown in Figure 1.

The thermostat has three locations. The first two, Heat and Cool, represent states in which the thermostat heats and cools the environment it operates in, respectively. The third, Check, represents a self-diagnostic state in which the thermostat does not heat or cool. The continuous state space of the thermostat consists of two continuous variables denoting an internally resettable clock c and the temperature T in the environment in which the thermostat operates.

Each location has an associated *invariant* predicate defining the set of permitted values for the continuous variables while in that location. The invariants

for the thermostat are:

$$\mathrm{Inv}_{\mathsf{Heat}}(c,T) := T \leqslant 10 \wedge c \leqslant 3, \quad \mathrm{Inv}_{\mathsf{Cool}}(c,T) := T \geqslant 5, \quad \mathrm{Inv}_{\mathsf{Check}}(c,T) := c \leqslant 1.$$

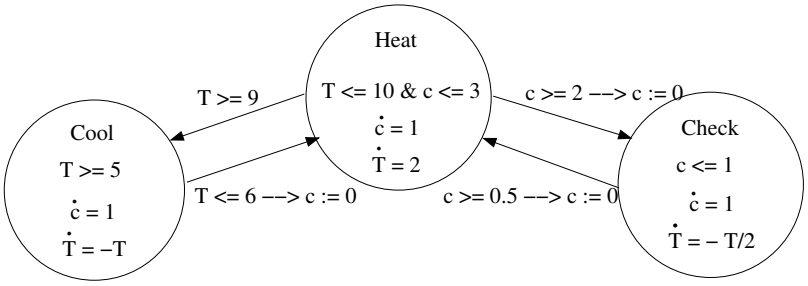

Fig. 1. The Thermostat as an example of a Hybrid Systems

The *initial states* of a hybrid system are determined by a predicate Init. For the thermostat, $\mathrm{Init}(l,c,T)$ is defined as $l = \mathsf{Heat} \wedge c = 0 \wedge 5 \leqslant T \leqslant 10$.

The *discrete* transitions between locations describe the logic of the software system. Each such transition is comprised of two components: a *guard* predicate defining a subset of the continuous state space in which the transition is enabled (permitted), and a *reset* function describing an instantaneous change applied as a side effect of the transition, as seen in the following definition of the discrete transition relation:

$$(l,p) \to_D (l',p') := \mathrm{guard}_{l,l'}(p) \wedge \mathrm{reset}_{l,l'}(p) = p' \wedge \mathrm{Inv}_l(p) \wedge \mathrm{Inv}_{l'}(p')$$

It will be clear from Figure 1 what the guards and reset functions are. Note the inherent non-determinism in a Hybrid Systems specification: when in Cool, the system can jump to Heat whenever the temperature T is in the interval $[5,6]$.

Each location in a hybrid system has an accompanying *flow function* which describes how the continuous variables change over time while the system is in that location. The idea is that different locations correspond to different uses of actuators available to the software system, the effects of which are reflected in the flow function. In the thermostat example, the flow function corresponding to the Cool location has the temperature decrease over time. This is expressed via the differential equation $\dot{T} = -T$, which is the usual short hand for $T'(t) = -T(t)$, where $T'(t)$ denotes the derivative of the temperature function over time t.

In the canonical definition of hybrid systems, flow functions are specified as solutions to differential equations (or differential inclusions) describing the dynamics of the continuous variables. We follow [2] in abstracting from these, taking instead the solutions of these differential equations, which are flow functions Φ which satisfy:

$$\Phi(p,0) = p \quad \text{and} \quad \Phi(p,d+d') = \Phi(\Phi(p,d),d')$$

The idea is that $\Phi(p,d)$ denotes the value of the continuous variable after duration d, starting from the value p. We say that there is a (concrete) *continuous transition* from a state (l,p) to a state (l,p') if there is a non-negative duration d such that $p' = \Phi_l(p,d)$ with the invariant for l holding at every point along the way:

$$(l,p) \to_C (l,p') := \exists_{d \in \mathbf{R}_{\geqslant 0}} . \, \Phi_l(p,d) = p' \wedge \forall_{0 \leqslant t \leqslant d} . \, \mathrm{Inv}_l(\Phi_l(p,t)).$$

A flow function on $\mathbb{R}^2$ can be expressed as the product of two flow functions: $\Phi_l((c_0,T_0),t) = (\varphi_{l,\mathsf{c}}((c_0,T_0),t), \varphi_{l,\mathsf{T}}((c_0,T_0),t))$. In the thermostat example, as in many other examples of hybrid systems, $\varphi_{l,\mathsf{c}}((c_0,T_0),t)$ does not actually depend on T_0 and $\varphi_{l,\mathsf{T}}((c_0,T_0),t)$ does not actually depend on c_0. We call this feature *separability* of the flow function. Our development currently relies heavily on this property. Separability makes the form of the flow functions simpler:

$$\Phi_l((c_0,T_0),t) = (\varphi_{l,\mathsf{c}}(c_0,t), \varphi_{l,\mathsf{T}}(T_0,t))$$

In the thermostat, $\varphi_{l,\mathsf{c}}(c_0,t) = c_0 + t$ for all locations l, $\varphi_{\mathsf{Heat},\mathsf{T}}(T_0,t) = T_0 + 2t$, $\varphi_{\mathsf{Check},\mathsf{T}}(T_0,t) = T_0 * e^{-\frac{1}{2}t}$ and $\varphi_{\mathsf{Cool},\mathsf{T}}(T_0,t) = T_0 * e^{-t}$. So $\varphi'_{\mathsf{Cool},\mathsf{T}}(T_0,t) = -\varphi_{\mathsf{Cool},\mathsf{T}}(T_0,t)$, solving the differential equation $\dot{T} = -T$ for the Cool location.

A transition is either continuous or discrete: $\to_{CD} := \to_D \cup \to_C$. A finite sequence of transitions constitutes a *trace* and we denote by $\twoheadrightarrow_{CD}$ the transitive reflexive closure of $\to_{CD}$. We now say that a state s is *reachable* if there is an initial state i from which there is a trace to s, that is

$$\mathrm{Reach}(s) := \exists_{i \in \mathrm{State}} . \, \mathrm{Init}(i) \wedge i \twoheadrightarrow_{CD} s.$$

The objective of hybrid system safety verification is to show that the set of reachable states is a subset of a predefined set of "safe" states. For the thermostat, the intent is to keep the temperature above 4.5 degrees at all times, and so we define $\mathrm{Safe}(c,T) := T > 4.5$ (and $\mathrm{Unsafe}(c,T)$ as its complement).

2.1 The Abstraction Method

There are uncountably many traces in a hybrid system, so safety is undecidable in general. In concrete cases, however, safety may be (easily) provable if one finds the proper *proof invariant*. Unfortunately these are often hard to find, so we prefer methods that are more easily automated. The *predicate abstraction* method of [2] is one such method.

The idea is to divide the continuous state space into a finite number of convex subsets (polygons), $A_1, \ldots, A_n$, which yields a finite *abstract state space*, $\mathrm{AState} := \{(l, A_i) \mid l \in \mathrm{Loc}, 1 \leqslant i \leqslant n\}$, with an obvious embedding $A : \mathrm{State} \to \mathrm{AState}$ of concrete states into abstract states. On this abstract state space, one immediately defines *abstract continuous transitions* and *abstract discrete transitions* (both potentially undecidable) as follows.

$$(l,P) \overset{A}{\to}_C (l,Q) := \exists_{p \in P, q \in Q} . \, (l,p) \to_C (l,q)$$
$$(l,P) \overset{A}{\to}_D (l',Q) := \exists_{p \in P, q \in Q} . \, (l,p) \to_D (l,q).$$

Define *abstract reachability* by $\mathrm{AReach}(a) := \exists_{s_0 \in \mathrm{State}} . \mathrm{Init}(s_0) \wedge A(s_0) \overset{A}{\rightarrow\!\!\!\!\rightarrow}_{CD} a$, as expected. Also the predicates ASafe and AUnsafe, stating when abstract states are safe / unsafe can be defined in the straightforward way.

Traces in the finite transition system constructed in this way are sufficiently representative (see Figure 2.) of those in the original (concrete) system that one can conclude safety of the latter from safety of the abstract system:

$$\text{if } \forall_{a \in \mathrm{AState}} . \mathrm{AReach}(a) \rightarrow \mathrm{ASafe}(a), \text{then } \forall_{s \in \mathrm{State}} . \mathrm{Reach}(s) \rightarrow \mathrm{Safe}(s).$$

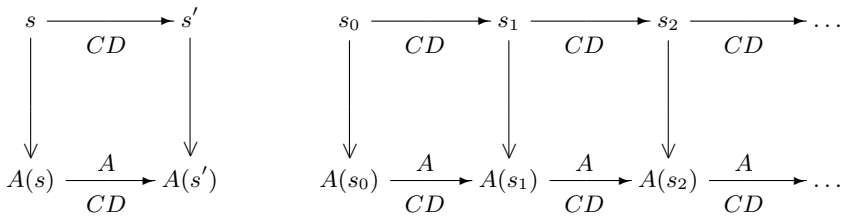

Fig. 2. The abstraction function

The interest and power of the abstraction method lies in two facts. First, we do not need the *exact* definitions of $\overset{A}{\rightarrow}_C$ and $\overset{A}{\rightarrow}_D$ to conclude safety of the concrete system from safety of the abstract system. We only need the property of Figure 2, so we can *over-estimate* $\overset{A}{\rightarrow}_C$ and $\overset{A}{\rightarrow}_D$ (i.e. replace it with a transition relation that allows more transitions). Second, there are good heuristics for how to divide the continuous state space into regions, and how to decide whether there should be an abstract transition from one abstract state to another.

This is indicated in Figure 3. The left hand side illustrates the challenge: given abstract regions A and B, we are to determine whether some flow duration permits flow from points in A to points in B. Following the over-estimation property just mentioned, we introduce an abstract transition from A to B whenever we cannot positively rule this out.

On the right hand side we see the abstract state space indicated for the location Heat. The abstract state space consists of rectangles, possibly degenerate (extending to $-\infty$ or $+\infty$). According to [2], a good candidate for an abstraction is to take the values occurring in the specification (Figure 1) as the bounds of such rectangles. (In case one cannot prove safety, there is of course the opportunity for the user to refine the bounds.) The grey area indicates that from these states also abstract discrete transitions are possible. The dashed area is unreachable, because of the invariant for the Heat location. being reachable. All the abstract transitions from the rectangle $[0.5, 1) \times [5, 6)$ are shown: as the temperature flow function for Heat is $\varphi_{\mathsf{Heat},\mathsf{T}}(T_0, t) = T_0 + 2 * t$, and the clock flow function is $\varphi_{\mathsf{Heat},\mathsf{c}}(c_0, t) = c_0 + t$, these are all the possible abstract transitions.

Using the abstraction method, [2] proves the correctness of the thermostat.

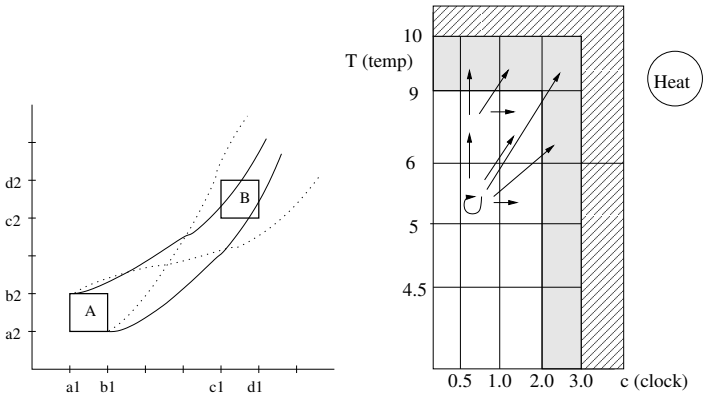

Fig. 3. The abstraction function computed

3 Formalisation

We now describe the Coq formalisation and the design choices made. We will not pay much attention to the specifics of Coq and its type theory CiC, and will instead focus on concerns relating to the use of computable reals and constructive logic. The complete sources of the development are available on the web, as well as a technical report describing the formalisation in more detail [22].

3.1 (Concrete) Hybrid Systems

We begin by showing our definition of a concrete system, the different parts of which we discuss in the remainder of this section.

Record $System : Type :=$
$\{Point : CSetoid$
$; Location : Set$
$; Location\_eq\_dec : EqDec\ Location\ eq$
$; locations : ExhaustiveList\ Location$
$; State := Location\ \times\ Point$
$; initial : State \to Prop$
$; invariant : State \to Prop$
$; invariant\_initial : initial\ subsetof\ invariant$
$; invariant\_stable : \forall\ s, Stable\ (invariant\ s)$
$; flow : Location \to Flow\ Point$
$; guard : State \to Location \to Prop$
$; reset : Location \to Location \to Point \to Point\}.$

This is a Coq definition of a record type of "Systems", which contain a field "Point", representing the continuous state space and a field "Location", representing the set of locations. Here, we take Point to be an arbitrary constructive setoid, which is basically just a type with an equality on it. For Location, we assume a decidable equality and a finite enumeration ("exhaustive list") of locations. The other parts are as expected ("Flow Point" is the type of flow functions on the type "Point"), except for the requirement that the invariant Inv_l is "Stable" for every location l, which we will discuss now.

3.2 Stability, Double Negation, and Computable Reals

Constructively, a proof of $X \to A \vee B$ is a function that, given an X, returns either a proof of A, or a proof of B. With this in mind, suppose we try to implement $le\_lt\_dec : \forall\ (x, y : CR), (x \leqslant y \vee y < x)$, where CR are the constructive reals. Then given x and y in CR, we are to produce a proof either of $x \leqslant y$ or of $y < x$. Unfortunately, the nature of computable reals only lets us observe arbitrarily close approximations of x and y. If it happens to be the case that $x = y$, then no matter how closely we approximate x and y, the error margins (however small) will always leave open the possibility that y is really smaller than x. Consequently, we will never be able to definitively conclude that $x \leqslant y$.

Computable reals do admit two variations of the proposition:

1. $leltdec_{overlap} : \forall\ (x\ y : CR), (x < y \to \forall\ z, (z \leqslant y \vee x \leqslant z))$
2. $leltdec_{DN} : \forall\ (x\ y : CR), \neg\neg(x \leqslant y \vee y < x)$

Both are weaker than the original, and are less straightforward to use. Nevertheless, this is the path we will take in our development (we will heavily use $leltdec_{DN}$), because just taking le_lt_dec as an axiom amounts to cheating. A question that immediately arises is: How does one actually use this doubly negated variant in proofs? One practical way is to observe that double negation, as a function on propositions, is a monad [21]. Writing $DN\ P$ for $\neg\neg P$, we have the following two key operations that make DN a monad:

$$return_{DN} : \forall\ A, A \to DN\ A$$
$$bind_{DN} : \forall\ A\ B, DN\ A \to (A \to DN\ B) \to DN\ B$$

The first expresses that any previously obtained result can always be inserted "into" the monad. The second expresses that results inside the monad may be used freely in proofs of additional properties in the monad. For instance, one may $bind_{DN}$ a proof of $DN(x \leqslant y \vee y < x)$ (obtained from $leltdec_{DN}$ above) with a proof of $(x \leqslant y \vee y < x) \to DNP$, yielding a proof of $DN\ P$.

Thus, DN establishes a "proving context" in which one may make use of lemmas yielding results inside DN that may not hold outside of it (such as $leltdec_{DN}$), as well as lemmas yielding results not in DN, which can always be injected into DN using $return_{DN}$. The catch is that such proofs always end up with results in DN, which begs the question: what good is any of this? In particular, can $leltdec_{DN}$ be used to prove anything not doubly negated?

As it happens, some propositions are *stable* in the sense that they are constructively equivalent to their own double negation. Examples include negations, non-strict inequalities on real numbers, and any decidable proposition.

Requiring that hybrid system invariants are stable effectively lets us use classical reasoning when showing that invariants hold in certain states. One instance where we need this is in the proof of transitivity of the concrete continuous transition.

Invariants are typically conjunctions of inequalities, which are stable only if the inequalities are non-strict. Hence, the limits on observability of computable real numbers ultimately mean that our development cannot cope with hybrid systems whose location invariants use strict inequalities. We feel that this is not a terrible loss. In Section 3.3 we will see analogous limitations in the choice of abstraction parameters.

3.3 Abstract Hybrid Systems

We now want to define an abstract system and an abstraction function satisfying the properties indicated in Figure 2. However, this is not possible, because we cannot make a case distinction like $x \leqslant 0 \vee 0 < x$ and therefore we cannot define a function that maps a point (c, T) to the rectangle R it is in. We can define a function that approximates a point (c, T) up to, say ϵ ($\epsilon > 0$) and then decides to send that point to the rectangle R the approximation is in. This implies that, when one is close to the edge of a rectangle,

- different *representations* of a point (c, T) may be sent to different rectangles,
- a point that is less than ϵ outside the rectangle R may still be sent to R.

The second is very problematic, because it means the property for the abstraction function A depicted in Figure 2 no longer holds.

We argue that these problems are not merely inconvenient byproducts of our use of constructive logic and computable reals, but actually reflect the profound limitation of physical reality where one can only ever measure quantities approximately, making case distinctions like $x \leqslant 0 \vee 0 < x$ simply unrealistic.

Moreover, we claim that the classical abstraction method allows one to prove the safety of systems that are unreliable in practice. We will not expand on this here, but suppose we add a fourth location Off to the thermostat of Figure 1, with $\dot{T} = -1$, $\dot{c} = 1$ and an arrow from Heat to Off with guard $c \geqslant 2 \wedge T < 9$. Clearly, if the system can end up in location Off, it is unsafe. Now, using the classical abstraction method, there is no transition to any state involving location Off from the initial state, because as soon as $c \geqslant 2$, $T \geqslant 9$. However, when we get close to $c = 2$, any small mistake in the measurement of T may send the system to Off, making the whole hybrid system very unreliable.

The positive thing is that we do not really need the commutation property of Figure 2. To address the problems we

- let regions in the abstract hybrid systems overlap (ideally as little as possible, e.g. only at the edges).

– replace the abstract relations $\overset{A}{\to}_C$ and $\overset{A}{\to}_D$ by functions over_cont_trans and over_disc_trans that take a region R_0 as input and output a *list of regions* including R_0: $(R_o, R_1, \ldots, R_n)$ in such a way that $\cup_{0 \leqslant i \leqslant n} R_i$ is an over-approximation of the set of states reachable by a continuous (resp. discrete) step from a state in R_0.

– loosen the requirement on the abstraction function A; for $s \in \text{State}$, we only require $DN(\exists_{r \in \text{Region}} . s \in r)$.

To summarise, if $s \to_C s'$, then we don't require $A(s')$ to be in the list over_cont_trans$(A(s))$, but we only require s' to be in the $\bigcup$over_cont_trans$(A(s))$. This simple change relieves us from having to determine the exact regions that points are in: they just should be covered. The functions over_cont_trans and over_disc_trans yield a notion of *trace* in the abstract hybrid system in the straightforward way: starting from R_0, take an R_1 in over_cont_trans(R_0), then an R_2 in over_disc_trans(R_1), and so forth.

Whereas in a concrete hybrid system states consist of a location paired with a point in the continuous state space, in an abstract hybrid system states consist of a location paired with the "name" of a region corresponding to a subset of the continuous state space. From now on we will use a "*concrete.*" prefix for names like *State* defined in section 3.1, which now have abstract counterparts. *Region* is a field from a record type *Space* bundling region sets with related requirements:

Record *Space* : *Type* :=
  {*Region* : *Set*
  ; *Region_eq_dec* : *EqDec Region eq*
  ; *regions* : *ExhaustiveList Region*
  ; *NoDup_regions* : *NoDup regions*
  ; *in_region* : *Container Point Region*
  ; *regions_cover* : ∀ (*l* : *Location*) (*p* : *Point*),
    invariant (*l*, *p*) → *DN* {*r* : *Region* | *p* ∈ *r*}}.

The *Container Point Region* type specified for *in_region* reduces to *Point* → *Region* → *Prop*. *Container* is a type class that provides the notation "$x \in y$". Finally, *regions_cover* expresses that each concrete point belonging to a valid state must be represented by a region—a crucial ingredient when arguing that unreachability in the abstract system implies unreachability in the concrete system. The double negation in its result type is both necessary and sufficient:

It is *necessary* because *regions_cover* boils down to a (partial) function that, given a concrete point, must select an abstract region containing that point. This means that it must be able to decide on which side of a border between two regions the given point lies. As we saw in section 3.2, that kind of decidability is only available inside *DN* unless all region borders have nontrivial overlap, which is undesirable.

Fortunately, the double negation is also *sufficient*, because we will ultimately only use *regions_cover* in a proof of ... → ¬*concrete.reachable s* (for some universally quantified variable *s*), which, due to its head type being a negation, is

stable, and can therefore be proved in and then extracted from DN. Hence, we only need *regions_cover*'s result in DN.

3.4 Under-Estimation and Over-Estimation

Ultimately, in our development we are writing a program that *attempts* to produce hybrid system safety proofs. Importantly, we are *not* writing a complete hybrid system safety decision procedure: if the concrete system is unsafe or the abstraction method fails, our program will simply not produce a safety proof. It might seem, then, that we are basically writing a *tactic* for a particular problem domain. However, tactics in Coq are normally written in a language called Ltac, and typically rely on things like pattern matching on syntax. Our development, on the other hand, is very much written in regular Gallina, with hardly any significant use of Ltac.

We define *underestimation* P to be either a proof of P, or not:

$$\textbf{Definition}\ \mathit{underestimation}\ (P : \mathit{Prop}) : \mathit{Set} := \{b : \mathit{bool} \mid b = \mathit{true} \to P\}.$$

The *bool* in the definition nicely illustrates why we call this an "under-estimation": it may be *false* even when P holds. We can now describe the functionality of our program by saying that it under-estimates hybrid system safety, yielding a term of type *underestimation Safe*, where *Safe* is a proposition expressing safety of a hybrid system.

Considered as theorems, under-estimations are not very interesting, because they can be trivially "proved" by taking *false*. Hence, the value of our program is not witnessed by the mere fact that it manages to produce terms of type *underestimation Safe*, but rather by the fact that when run, it actually manages to return *true* for the hybrid system we are interested in (e.g. the thermostat). It is for this reason that we primarily think of the development as a program rather than a proof, even though the program's purpose is to produce proofs.

The opposite of an under-estimation is an over-estimation:

$$\textbf{Definition}\ \mathit{overestimation}\ (P : \mathit{Prop}) : \mathit{Set} := \{b : \mathit{bool} \mid b = \mathit{false} \to \neg P\}.$$

Since hybrid system safety is defined as unreachability of unsafe states, we may equivalently express the functionality of our development by saying that it over-estimates unsafe state reachability. Indeed, most subroutines in our programs will be over-estimators rather than under-estimators. Notions of over-estimation and under-estimation trickle down through all layers of our development, down to basic arithmetic. For instance, we employ functions such as (recall that CR denotes the type of constructive reals):

$$\mathit{overestimate}_{\leqslant_{CR}}\ (\epsilon : \mathbb{Q}^+) : \forall\ x\ y : CR, \mathit{overestimation}\ (x \leqslant_{CR} y)$$

As discussed earlier, $\leqslant_{CR}$ is not decidable. $\mathit{overestimate}_{\leqslant_{CR}}$ merely makes a "best effort" to prove $\neg(x \leqslant_{CR} y)$ using ϵ-approximations. A smaller ϵ will result in fewer spurious *true* results.

3.5 Abstract Space Construction

When building an abstract system, one is in principle free to divide the continuous state space up whichever way one likes. However, if the regions are too fine-grained, there will have to be very many of them to cover the continuous state space of the concrete system, resulting in poor performance. On the other hand, if the regions are too coarse, they will fail to capture the subtleties of the hybrid system that allow to prove it safe (if indeed it is safe at all). Furthermore, careless use of region overlap can result in undesirable abstract transitions (and therefore traces), adversely affecting the abstract system's utility.

In [2], a heuristic for interval bound selection is described, where the bounds are taken from the constants that occur in the invariant, guard, and safety predicates. For the thermostat, we initially attempted to follow this heuristic and use the same bounds, but found that due to our use of computable reals, we had to tweak the bounds somewhat to let the system successfully produce a safety proof. Having to do this "tweaking" manually is clearly not ideal. One may want to develop heuristics for this.

Another way in which our thermostat regions differ from [2] lies in the fact that our bounds are always inclusive, which means adjacent regions overlap in lines.

3.6 Abstract Transitions and Reachability

Once we have a satisfactory abstract *Space*, our goal is to construct an over-estimatable notion of abstract reachability implied by concrete reachability, so that concrete unreachability results may be obtained simply by executing the abstract reachability over-estimator. We first over-estimate the continuous transitions; we need the following definition for that.

Definition *shared_cover*
$(cs : concrete.State \to Prop)\ (ss : abstract.State \to Prop) : Prop :=$
$\forall\ s : concrete.State, s \in cs \to DN\ (\exists\ r : abstract.State, s \in r \wedge r \in ss).$

A set of concrete states is said to be sharedly-covered by a set of abstract states if for each of the concrete states in the former there is an abstract state in the latter that contains it.

We now specify what the type of *over_cont_trans* should be.

$over\_cont\_trans : \forall\ s : abstract.State,$
$\{p : list\ abstract.State \mid NoDup\ p \wedge shared\_cover$
$(concrete.invariant \cap (overlap\ s \circ flip\ concrete.cont\_trans))\ p\}$

So, *over_cont_trans* s should produce a list of abstract states p without duplicates, such that p is a shared cover of the collection of concrete states c that satisfy the invariant and whose set of origins under *concrete.cont_trans* have an overlap with s. In more mathematical terms: p should form a shared cover of $\{c \in \text{State} \mid \text{Inv}(c) \wedge s \cap \{c' \mid c' \to_C c\} \neq \emptyset\}$. We similarly specify *over_disc_trans*

as an over-estimator for *concrete_disc_trans* and *over_initial* as an over-estimator of *concrete.initial*.

We now consider the properties we require for *abstract.reachable*. An obvious candidate is:

$$\forall\ (s : concrete.State), concrete.reachable\ s \rightarrow \\ \quad \forall\ (s' : abstract.State), s \in s' \rightarrow abstract.reachable\ s'.$$

because it implies

$$\forall\ (s : concrete.State) \\ \quad (\exists\ s' : abstract.State, s \in s' \wedge \neg abstract.reachable\ s') \rightarrow \\ \quad\quad \neg concrete.reachable\ s,$$

This expresses that to conclude unreachability of a concrete state, one only needs to establish unreachability of *one* abstract state that contains it. However, this definition neglects to facilitate *sharing*: a concrete state may be in more than one abstract state. So, if a concrete state is in one abstract state which is unreachable, it may still be in another abstract state which *is* reachable. One should establish unreachability of *all* abstract states containing the concrete state. Hence, what we really want is an *abstract.reachable* satisfying:

$$\forall\ s : concrete.State, \\ \quad (\forall\ s' : abstract.State, s \in s' \rightarrow \neg abstract.reachable\ s') \rightarrow \\ \quad \neg concrete.reachable\ s.$$

3.7 Under-Estimating Safety

We now show how a decision procedure for *abstract.reachable* lets us under-estimate hybrid system safety, and in particular, lets us obtain a proof of thermostat safety. (The construction of the decision procedure itself is detailed in the next section.) So suppose we have $reachable\_dec : decider\ abstract.reachable$. *ThermoSafe* is defined as $thermo\_unsafe \subseteq concrete.unreachable$. Since we trivially have $\neg overlap\ unsafe\ concrete.reachable \rightarrow ThermoSafe$, we also have:

Definition $under\_thermo\_unsafe\_unreachable : underestimation\ ThermoSafe$.

Using a tiny utility $underestimation\_true$ of type $\forall\ P\ (o{:}underestimation\ P), o = true \rightarrow P$, we can now *run* this under-estimator to obtain a proof of the thermostat system's safety:

Theorem : *ThermoSafe*.
Proof.
apply (*underestimation_true under_thermo_unsafe_unreachable*).
vm_compute.reflexivity.
Qed.

The first *apply* reduces the goal to

$$under\_thermo\_unsafe\_unreachable = true.$$

The *vm_compute* tactic invocation then forces evaluation of the left hand side, which will in turn evaluate *over_thermo_unsafe_reachable*, which will evaluate *reachable_dec*, which will evaluate the over-estimators of the continuous and discrete transitions. This process, which takes about 35 seconds on a modern desktop machine, eventually reduces *under_thermo_unsafe_unreachable* to *true*, leaving $true = true$, proved by *reflexivity*.

We can now also clearly see what happens when the abstraction method "fails" due to poor region selection, overly simplistic transition/initiality over-estimators, or plain unsafety of the system. In all these cases, *vm_compute* reduces *under_thermo_unsafe_unreachable* to *false*, and the subsequent *reflexivity* invocation will fail.

This concludes the high level story of our development. What remains are the implementation of *reachable_dec* in terms of the decidable over-estimators for abstract initiality and continuous and discrete transitions, and the implementation of those over-estimators themselves. The former is a formally verified graph reachability algorithm, that we don't detail here. The over-estimator for continuous transitions, *over_cont_trans* will be detailed in the next section for the thermostat case.

3.8 Over-Estimating Continuous Abstract Transitions

We now discuss the implementation of *over_cont_trans*. Given two regions *r_src* and *r_dst*, if we can determine that there are no points in *r_src* which the flow function maps to points in *r_dst*, then we don't put an abstract continuous transition between *r_src* and *r_dst*. Clearly, this is impossible to meaningfully over-estimate for a general flow function and general regions. However, the thermostat posesses three key properties that we can exploit:

1. its continuous space is of the form $\mathbf{R}^n$;
2. abstract regions correspond to multiplied $\mathbf{R}$ intervals;
3. its flow functions are both *separable* and *range invertible*.

The notion of *separability* has already been discussed in Section 2.
A flow function f on CR is *range invertible* if

$$\begin{aligned}&\exists\ (range\_inverse : OpenRange \to OpenRange \to OpenRange),\\ &\quad \forall\ (a : OpenRange)\ (p : CR), p \in a \to\\ &\quad \forall\ (b : OpenRange)\ (d : Duration), f\ p\ d \in b \to d \in range\_inverse\ a\ b\end{aligned}$$

Here, *OpenRange* represents potentially unbounded intervals in $\mathbf{R}$ (with bounds closed if present. In other words, if $\varphi : \mathbf{R}^2 \to \mathbf{R}$ is a flow function with range inverse F and a, b are intervals in $\mathbf{R}$, then $F(a, b)$ is an interval that contains all t for which $\varphi(x, t) \in b$ for some $x \in a$. Range invertibility is a less

demanding alternative to point invertibility: φ^{-1} is the *point inverse* of φ if $\forall x, y \in \mathbf{R}(\varphi(x, \varphi^{-}1(x, y)) = y)$. So a point inverse $\varphi^{-1}(x, y)$ computes the exact time t it takes to go from x to y via flow φ. A range inverse computes an interval that contains this t.

In the formalisation we use a modest library of flow functions when defining the thermostat's flow. Included in that library are range-inverses, which consequently automatically apply to the thermostat's flow. Hence, no ad-hoc work is needed to show that the thermostat's flow functions are range-invertible. Having defined the class of separable range-invertible flow functions, and having argued that the thermostat's flow is in this class, we now show how to proceed with our over-estimation of existence of points in *r_src* which the flow function map to points in *r_dst*. Regions in the abstract space for our thermostat are basically pairs of regions in the composite spaces, so *r_src* and *r_dst* can be written as (*r_src_temp*, *r_src_clock*) and (*r_dst_temp*, *r_dst_clock*), respectively, where each of these four components are intervals.

We now simply use an *OpenRange* overlap over-estimator of type

$$\mathbb{Q}^{+} \to \forall\ a\ b : \mathit{OpenRange}, \mathit{overestimation}\ (\mathit{overlap}\ a\ b)$$

(defined in terms of things like $\mathit{overestimate}_{\leqslant_{CR}}$ shown in section 3.4) to over-estimate whether the following three intervals overlap:

1. $[0, \mathit{inf}]$
2. *range_inverse temp_flow r_src_temp r_dst_temp*
3. *range_inverse clock_flow r_src_clock r_dst_clock*

For a visual explanation, one may consult the left drawing in Figure 3 and view *r_src_clock* as $[a_1, b_1]$, *r_dst_clock* as $[c_1, d_1]$ etc. Overlap of 2 and 3 is equivalent to existence of a point in *r_src* from which one can flow to a point in *r_dst*. After all, if these two range inverses overlap, then there is a duration d that takes a certain temperature value in *r_src_temp* to a value in *r_dst_temp* and also takes a certain clock value in *r_src_clock* to a value in *r_dst_clock*. If 2 and 3 do *not* overlap, then either it takes so long for the temperature to flow from *r_src_temp* to *r_dst_temp* that any clock value in *r_src_clock* would "overshoot" *r_dst_clock*, or vice versa. Finally, if 1 does not overlap with 2 and 3, then apparently one could only flow backward in time, which is not permitted. Hence, overlap of these three ranges is a necessary condition for existence of concrete flow from points in *r_src* to points in *r_dst*, and so our *abstract.cont_trans* over-estimator may justifiably return "false" when the overlap over-estimator manages to prove absence of overlap.

4 Related work

Verification of hybrid systems is an active field of research and there is a number of tools developed with this goal in mind; see [15] for a comprehensive list. Most of them are based on abstract refinement methods, either using box representations [20,17] or with polyhedra approximations [2,5,6].

Many of those tools are implemented in MATLAB [13] and those using some general programming language of choice most often rely on standard floating point arithmetic, which comes with its rounding errors. Some tools that address this problem include PHAVer [8], which relies on the Parma Polyhedra Library [3] for exact computations with non-convex polyhedra and HSolver [19], which is based on the constraint solver RSolver [18].

Formal verification becomes more and more important, especially in the field of hybrid systems, which are used to model safety critical systems of growing complexity. There has been previous work on using general purpose theorem provers for verification of hybrid systems: see [1,10] and [12,4] for works using, respectively, PVS and STeP. KeYmaera [16] is a dedicated interactive theorem prover for specification and verification logic for hybrid systems. It combines deductive, real algebraic, and computer algebraic prover technologies and allows users to model hybrid systems, specify their properties and prove them in a semi-automated way.

However, to the best of our knowledge, none of the work or tools discussed above rely on a precise model of real number computations completely verified in a theorem prover, such as the model of CoRN used in this work.

5 Conclusions and further research

The presented verification of hybrid systems in Coq gives a nice showcase of proof-by-computation-on-computable-reals. The computable reals in CoRN do really complicated things for us, by approximating values for various real number expressions at great precision. The development also contains some nice layers of abstraction, involving the sophisticated use of type classes, e.g. the systematic use of estimators to make tactic-like optional-deciders, at each level in the stack and the use of the double negation monad.

It remains to be seen how far this automated verification approach can be taken, given the fact that we have limited ourselves to hybrid systems where we have a *solution* to the differential equation as a flow function, this flow function is *separable* (Section 2) and *range invertible* (Section 3.8) and the invariants are *stable* (Section 3.1). Finally, the reset functions should not be too strange, see [22] for details. If the flow functions are given, the largest part of the work is in producing *useful* range inverse functions. Note that we always have the trivial range inverse function, that just returns $\mathbf{R}$, but that is not useful. We want a function that actually helps to exclude certain abstract continuous transitions.

There is still a lot of room for more clever heuristics, possibly with less restrictive preconditions. The heuristic in [2] for bound selection doesn't work out of the box, but manual tweaking is obviously not ideal, so some more experimentation is required here. Finally, in case safety cannot be proved, one would like the system to generate an "offending trace" automatically, which can then be inspected by the user.

Acknowledgements We thank the referees for their useful suggestions; due to space limitations we have not been able to incorporate all of them.

References

1. E. Ábrahám-Mumm, U. Hannemann, and M. Steffen. Assertion-based analysis of hybrid systems with PVS. In *Proceedings of Computer Aided Systems Theory (EUROCAST '01)*, volume 2178 of *LNCS*, pages 94–109, 2001.
2. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *ACM Trans. Embedded Comput. Syst*, 5:152–199, 2006.
3. R. Bagnara, P. Hill, and E. Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.
4. N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. *Theor. Comput. Sci.*, 253(1):27–60, 2001.
5. A. Chutinan and B. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *HSCC '99*, volume 1569 of *LNCS*, pages 76–90, 1999.
6. E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.
7. L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *MKM*, volume 3119 of *LNCS*, pages 88–103, 2004.
8. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *HSCC '05*, volume 3414 of *LNCS*, pages 258–273, 2005.
9. T. Henzinger. The theory of hybrid automata. pages 278–292, 1996.
10. T. Henzinger and V. Rusu. Reachability verification for hybrid automata. In *HSCC '98*, volume 1386 of *LNCS*, pages 190–204, 1998.
11. N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.
12. Z. Manna and H. Sipma. Deductive verification of hybrid systems using STeP. In *HSCC '98*, volume 5195 of *LNCS*, pages 305–318, 1998.
13. The MathWorks. MATLAB. `http://www.mathworks.com/products/matlab/`.
14. R. O'Connor. Certified exact transcendental real number computation in Coq. In *TPHOLs '08*, volume 5170 of *LNCS*, pages 246–261, 2008.
15. G. Pappas. Hybrid system tools. `http://wiki.grasp.upenn.edu/hst/`.
16. A. Platzer and J.-D. Quesel. Keymaera: A hybrid theorem prover for hybrid systems. In *IJCAR '08*, volume 5195 of *LNCS*, pages 171–178, 2008.
17. J. Preußig, S. Kowalewski, H. Wong-Toi, and T. Henzinger. An algorithm for the approximative analysis of rectangular automata. In *FTRTFT '98*, volume 1486 of *LNCS*, pages 228–240, 1998.
18. S. Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.
19. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions in Embedded Computing Systems*, 6(1), 2007.
20. O. Stursberg, S. Kowalewski, I. Hoffmann, and J. Preußig. Comparing timed and hybrid automata as approximations of continuous systems. In *Hybrid Systems*, volume 1273 of *LNCS*, pages 361–377, 1996.
21. P. Wadler. Monads for functional programming. In J.Jeuring and E.Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52, 1995.
22. E. van der Weegen. Automated machine-checked hybrid system safety proofs, an implementation of the abstraction method in Coq. Technical report, Radboud University Nijmegen, 2009. `http://www.eelis.net/research/hybrid/`.