Automated Machine-Checked Hybrid System Safety Proofs

An Implementation of the Abstraction Method In Coq

Eelis van der Weegen

Institute for Computing and Information Sciences
Raboud University Nijmegen

Abstract. This technical report documents our development of a hybrid system safety prover, implemented in Coq using the abstraction method introduced by Alur in [1]. The development includes: a formalization of the structure of hybrid systems; a systematic approach and generic set of support utilities for the construction of an abstract system (consisting of decidable "overestimators" of abstract transitions and initiality) faithfully representing a (concrete) hybrid system; a translation of abstract systems to graphs enabling decision of abstract state reachability using a certified graph reachability algorithm; a proof of an example hybrid system (taken from [1]) generated using this tool stack. The development critically relies on the computable real number implementation part of the C-CoRN library of formalized constructive mathematics. [1]

[1] This research was supported by the BRICKS/FOCUS project 642.000.501 "Advancing the Real use of Proof Assistants".

Table of Contents

1 Introduction

In [1], Alur presents a method for automated hybrid system safety verification based on the construction of an *abstract* hybrid system (essentially a finite automaton) corresponding to the hybrid system of interest. The abstract system is constructed such that traces in the original system are represented in the abstract system. Consequently, one can draw conclusions about reachability of states in the concrete system from analysis of state reachability in the abstract system. Since the abstract system is an entirely finite discrete object (unlike the concrete system), reachability can simply be computed (using any graph reachability algorithm). Thus, the abstraction method brings the safety verification

problem from a continuous and infinite domain into a discrete and finite domain, where it is amenable to what amounts to brute force.

The prototype implementation described in [1] was developed in a conventional programming language using libraries that use ordinary floating point arithmetic . The potential for bugs and floating point artifacts inherent in this approach limits the confidence one can justifiably have in safety judgements made by such an implementation. In this report, we describe a reimplementation of the basic technique in Coq, a proof assistant based on a rich type theory that also functions as a programming language, letting one develop "certified" programs: programs whose correctness is proved formally within the system. In our case, the program will be one that produces machine-checked proofs of hybrid system safety, obtained using the abstraction method.

Our development relies crucially on computation using real numbers, for which we use the computable real number implementation developed by Russell O'Connor [**?**], and part of the C-CoRN library of formalized constructive mathematics [**?**]. Indeed, this development showcases its use in a concrete and practical application, and subtleties regarding the nature of these computable reals are reflected in this development in several ways, which we will discuss in some detail.

Organization Throughout this report, we present in parallel our general formalized framework (with particular emphasis on differences with Alur's presentation), and the way it is used to obtain a safety proof for an example hybrid system, taken from [1], modeling the operation of a thermostat. In section 2, we begin by defining the structure of normal (concrete) hybrid systems, their semantics, reachability, and safety. We also define the concrete thermostat and the safety condition we wish to prove. In section 3, we take a moment to discuss some of the limits on decidability of properties of computable real numbers, as these complicate matters in the remainder of the development. Next, in section 4 we describe notions of underestimation and overestimation that are applied throughout the development. Then, in section 5, we begin our presentation of the abstraction method by discussing abstract regions, states, and spaces. We build on these in section 6, where we first develop the "obvious" notions of abstract system and reachability, discuss why they fail, and then rework them to embrace reachability sharing. In section 7 we show how, assuming abstract reachability is decidable, we may prove hybrid system safety by computation. Then, in section 8, we show how an abstract system consisting of transition and initiality "overestimators" can be used to decide abstract reachability by construction of a directed graph corresponding to the system, and the use of an ordinary but verified graph reachability algorithm. Finally, in sections 9 and 10 we describe the implementation of the aforementioned overestimators. We end with conclusions in section 11.

2 (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* × *Point*
; *initial* : *State* → *Prop*
; *invariant* : *State* → *Prop*
; *invariant_initial* : *initial* ⊆ *invariant*
; *invariant_mor* : *Morphism* (*eq* ⟹ *cs_eq* ⟹ *iff*) (*curry invariant*)
; *invariant_stable* : ∀ *s*, *Stable* (*invariant s*)
; *flow* : *Location* → *Flow Point*
; *guard* : *State* → *Location* → *Prop*
; *reset* : *Location* → *Location* → *Point* → *Point*
}.

2.1 States

A hybrid system is a model of how a software system, described as a finite set of *locations* [2] 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).

In [1], Alur requires that the continuous state space be a convex polyhedron in $\mathbb{R}^n$. In our definition of a hybrid system, we initially let the continuous state space be an arbitrary (constructive) setoid, called *Point*. We also explicitly require decidability of location equality.

Our running example (taken from [1]) is a thermostat whose discrete component consists of three locations (called *Heat*, *Cool*, and *Check*), and whose continuous state space is $\mathbb{R}^2$, where $\mathbb{R}$ are the *computable* reals. The *Heat* and *Cool* locations represent states in which the thermostat heats and cools the environment it operates in, respectively. The *Check* state is a self-diagnostic state in which the thermostat does not heat or cool. The first $\mathbb{R}$ component in the continuous state space represents an internally resettable clock, while the second represents the temperature in the environment in which the thermostat operates.

A *State* in a hybrid system is a pair (l, p) consisting of a location l and a point p in the continuous state space (representing an assignment of the continuous variables).

[2] We reserve the term "state" for something else.

The *invariant* predicate defines, for each location, a set of permitted values for the continuous variables. We will use this in the definitions of transition relations in a moment. The morphism field expresses that this invariant respects *Point*'s setoid equality. We further require that it is *stable*, where *stable* P is defined as $\neg\neg P \to P$. We discuss the reason for this stability requirement in section 3.

For the thermostat, the invariant is as follows:

Definition *invariant* $(s : State) : Prop :=$
  $0 \leqslant clock\ s \wedge$
  match *location* s **with**
  $\mid Heat \Rightarrow temp\ s \leqslant 10 \wedge clock\ s \leqslant 3$
  $\mid Cool \Rightarrow 5 \leqslant temp\ s$
  $\mid Check \Rightarrow clock\ s \leqslant 1$
  end.

Associated with the hybrid system is a set of *initial states*. For our thermostat example, the initial states are

$$\{\, s : State \mid location\ s = Heat \wedge 5 \leqslant temp\ s \leqslant 10 \wedge clock\ s = 0 \,\}$$

where *location*, *temp*, and *clock* are the obvious projections. For the thermostat, the requirement that the invariant holds at each initial state is easily proved.

The remaining parts of a hybrid system (i.e. *flow*, *guard* and *reset*) describe transitions between states, which, together with the set of initial states, determine the set of *reachable* states, representing the possible behaviors exhibited by a hypothetical real-world implementation of the hybrid system (as software running on a device with sensors and actuators).

2.2 Flow and Continuous Transitions

Each location in a concrete 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 the different locations corresponds to different uses of actuators available to the software system, the effects of which are described by the flow function. For instance, in our thermostat, the flow function corresponding to the *Heat* location will have the temperature increase with time, modeling the effect of the heater component in our imagined thermostat device.

In the canonical definition of hybrid systems, flow functions are specified as solutions to differential equations describing the dynamics of the continuous variables. We follow Alur's example in abstracting from these, taking instead functions ϕ of type $Point \to \mathbb{R}_{\geq 0} \to Point$ which satisfy:

$$\phi\ p\ 0 = p$$
$$\phi\ p\ (t + t') = \phi\ (\phi\ t\ p)\ t'$$

We further require that ϕ is a morphism respecting *Point*'s setoid equality. We bundle functions with these properties as a record type called *Flow* (not shown),

which is parameterized over the continuous state space type, and is equipped with an implicit coercion to the ϕ function it contains.

We now say that there is a (concrete) *continuous transition* from a state (l, p) to a state (l', p') if $l = l'$ and there is a non-negative duration d such that $p' = \mathit{flow}\ l\ p\ d$ with the invariant for l holding at every point along the way:

Definition *can_flow* $(l : \mathit{Location}) : \mathit{relation}\ \mathit{Point}$
$:= \lambda p\ p' \Rightarrow \exists\ d : \mathbb{R}_{\geq 0}, \mathit{flow}\ l\ p\ d = p' \wedge$
$\forall\ t, 0 \leqslant t \leqslant d \rightarrow \mathit{invariant}\ (l, \mathit{flow}\ l\ p\ t).$

Definition *cont_trans* : *relation State*
$:= \lambda(l, p)\ (l', p') \Rightarrow l = l' \wedge \mathit{can\_flow}\ l\ p\ p'.$

In our thermostat example, we express the flow function as the product of two flow functions on $\mathbb{R}$:

Definition *thermo_flow* $(l : \mathit{Location}) : \mathit{Flow}\ \mathbb{R}^2$
$:= \mathit{product\_flow}\ (\mathit{clock\_flow}\ l)\ (\mathit{temp\_flow}\ l).$

Here, *product_flow* has type $\forall\ X\ Y, \mathit{Flow}\ X \rightarrow \mathit{Flow}\ Y \rightarrow \mathit{Flow}\ (X \times Y)$, and includes the trivial proofs showing that flow in a product space can be formed by combining flows in the respective components.

Intuitively, *clock_flow* and *temp_flow* are the functions

Definition *clock_flow* $(l : \mathit{Location})\ (c : \mathbb{R})\ (d : \mathbb{R}_{\geq 0}) : \mathbb{R} :=$
$c + d.$

Definition *temp_flow* $(l : \mathit{Location})\ (t : \mathbb{R})\ (d : \mathbb{R}_{\geq 0}) : \mathbb{R} :=$
match l **with**
$\mid \mathit{Heat} \Rightarrow t + 2 * d$
$\mid \mathit{Cool} \Rightarrow t * \mathit{exp}\ (-d)$
$\mid \mathit{Check} \Rightarrow t * \mathit{exp}\ (-\frac{1}{2} * d)$
end.

However, to define the system, we actually require *Flow* records carrying additional proofs that these functions are proper flow functions. While these proofs could be given in an ad-hoc fashion for this thermostat system, we have instead developed a modest library of reusable flow functions and adaptors that let one compose flow functions that are proper by construction. Using these, *clock_flow* and *temp_flow* are defined as:

Definition *clock_flow* $(l : \mathit{Location}) : \mathit{Flow}\ \mathbb{R} :=$
flow.positive_linear

Definition *temp_flow* $(l : \mathit{Location}) : \mathit{Flow}\ \mathbb{R} :=$
match l **with**
$\mid \mathit{Heat} \Rightarrow \mathit{flow.scale}\ 2\ \mathit{flow.positive\_linear}$
$\mid \mathit{Cool} \Rightarrow \mathit{flow.decreasing\_exponential}$
$\mid \mathit{Check} \Rightarrow \mathit{flow.scale}\ (\frac{1}{2})\ \mathit{flow.decreasing\_exponential}$
end.

2.3 Discrete Transitions

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

```
Definition disc_trans : relation State := λ(l, p) (l', p') ⇒
  guard (l, p) l' ∧ reset l l' p = p' ∧
  invariant (l, p) ∧ invariant (l', p').
```

The thermostat we are modeling has four transitions, as expressed by the following guard:

```
Definition thermo_guard (s : State) (l : Location) : Prop :=
  match location s, l with
  | Heat, Cool ⇒ 9 ⩽ temp s
  | Cool, Heat ⇒ temp s ⩽ 6
  | Heat, Check ⇒ 2 ⩽ clock s
  | Check, Heat ⇒ 1/2 ⩽ clock s
  | _, _ ⇒ ⊥
  end.
```

Our reset function resets the clock for all but one of these transitions, and leaves the temperature variable as is:

```
Definition thermo_reset (l l' : Location) (p : Point) : Point :=
  (match l, l' with
     | Cool, Heat | Heat, Check | Check, Heat ⇒ 0
     | _, _ ⇒ fst p
   end
  , snd p).
```

Here we can discern a conceptual distinction between continuous variables directly controlled by the system (such as the clock in our thermostat), and variables that model a physical phenomenon (such as the temperature in our thermostat). This distinction is not made explicit in the definition of a hybrid system; in principle, nothing is stopping the thermostat from treating temperature as a variable of the former kind and resetting it to whatever value it pleases. However, this would simply make the system unimplementable.

2.4 Traces and Safety

A transition is either continuous or discrete:

Definition *trans* : *relation State* := *disc_trans* ∪ *cont_trans*.

We now say that a state s is *reachable* if there is an initial state i from which one can, by a finite number of transitions, end up in s:

Definition *reachable* $(s : State) : Prop :=$
$\exists\, i : State, initial\ i \wedge trans^*\ i\ s.$

Here, *trans** is the transitive reflexive closure of *trans*.

As mentioned before, the set of reachable states represents the possible behaviors exhibited by a hypothetical real-world implementation of the hybrid system (as software running on a device with sensor and actuators).

The idea now is that the purpose of a hybrid system is typically to keep the continuous variables within certain limits. In other words, to limit the set of reachable states to some "safe" subset of the complete state space. For our thermostat example, the intent is to keep the temperature above 4.5 degrees at all times, and so the safe states are defined to be those in which the temperature component is >4.5:

Definition *unsafe_thermo_state* : *Ensemble State* :=
$\lambda s \Rightarrow temp\ s \leqslant 4.5.$

(An *Ensemble* T is just a $T \to Prop$.)

The goal, then, is to verify that the reachable states are a subset of the safe states. Hence, for the thermostat, our main theorem is the following:

Theorem *ThermoSafe* : *unsafe_thermo_state* ⊆ *unreachable*.

"$A \subseteq B$" is just notation for "$\forall\ x, A\ x \to B\ x$", and *unreachable* is merely the complement of *reachable*.

Since we required that the invariant held at each initial state, and further defined the continous and discrete transitions such that the invariant had to hold everywhere along the path, a simple induction proof shows that

reachable ⊆ *invariant*.

Before we continue with our presentation of the abstraction method, we first take a moment to discuss some of the limits on decidability of properties of computable real numbers, as their consequences can be seen in many places in the development. Indeed, we already saw one such instance in the stability requirement for invariants.

3 Double Negation and Stability

One obvious and useful property we can derive is transitivity of the continuous transition relation, the proof of which reveals the need for invariant stability.

Suppose we have *cont_trans* (l, p) (l', p') and *cont_trans* (l', p') (l'', p'') for locations l, l', l'' and points p, p', p''. To show *cont_trans* (l, p) (l'', p''), we must show two things. The first, $l = l''$, follows immediately from transitivity of $=$. For the second, *can_flow* l p p'', we simply take the flow duration to be the sum of the durations from p to p' and from p' to p'' (call these d and d', respectively), and observe:

$$\begin{aligned} \mathit{flow}\ l\ p\ (d + d') &= \mathit{flow}\ l\ (\mathit{flow}\ l\ p\ d)\ d' \\ &= \mathit{flow}\ l\ p'\ d' \\ &= p'' \end{aligned}$$

What remains is to show that the invariant holds at each point along the way. That is,

$$\forall\ t, 0 \leqslant t \leqslant d + d' \rightarrow \mathit{invariant}\ (l, \mathit{flow}\ l\ p\ t).$$

From *can_flow* p p' we know this is true for $0 \leqslant t \leqslant d$, and from *can_flow* p' p'' we know that this is true for $d \leqslant t \leqslant d + d'$. Classically, then, the proof is a done deal, for one can simply distinguish cases $t \leqslant d$ and $d < t$.

Unfortunately, such case distinction is a luxury we do not have, because for computable reals, the proposition

$$\mathit{le\_lt\_dec} : \forall\ x\ y : \mathbb{R}, x \leqslant y \vee y < x.$$

is not provable. After all, to have a constructive proof of a disjunction $A \vee B$ (where A and B are arbitrary propositions/types) is to have either a proof of A or a proof of B. Similarly, to have a proof of $X \rightarrow A \vee B$ is to have a function that, given an X, either returns a proof of A, or a proof of B. With this in mind, suppose we try to implement *le_lt_dec*. Then given x and y in $\mathbb{R}$, 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. Now suppose $x = y$. 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 on the proposition:

1. $\mathit{le\_lt\_dec}_{overlap} : \forall\ x\ y : R, x < y \rightarrow \forall\ z, z \leqslant y \vee x \leqslant z$
2. $\mathit{le\_lt\_dec}_{DN} : \forall\ x\ y : R, \neg\neg(x \leqslant y \vee y < x)$

Both are weaker than the original, and are less straightforward to use. Nevertheless, this is the path we will take in our development, partly out of necessity (because in some parts of the development, we really need to "run" these lemmas to obtain $\leqslant/<$ proofs), and partly because just taking *le_lt_dec* as an axiom amounts to cheating.

For our transitivity proof, we will use the variant expressed using double negation. Two questions immediately arise when considering this variant. First,

why does the double negation make it provable? And second, how does one actually use this doubly negated variant in proofs?

For the first, we need only observe that $x \leqslant y$ is equivalent to $y \not< x$, and that the law of the excluded middle holds under double negation. That is,

$$\forall\ P, \neg\neg(P \vee \neg P)$$

is a trivial tautology considering that $\neg P$ is taken to mean $P \to \bot$.

One practical way to answer the question of how such a doubly negated proposition might be used to prove things is to observe that double negation, as a function on types/propositions, is a monad [4]. Writing $DN\ P$ for $\neg\neg P$, we have the following two key operations that make DN a monad:

$$\mathit{return}_{DN} : \forall\ A, A \to DN\ A$$
$$\mathit{bind}_{DN} : \forall\ A\ B, DN\ A \to (A \to DN\ B) \to DN\ B$$

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 $\mathit{le\_lt\_dec}_{DN}$ above) with a proof of $(x \leqslant y \vee y < x) \to DN\ P$, 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 $\mathit{le\_lt\_dec}_{DN}$), 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 $\mathit{le\_lt\_dec}_{DN}$ be used to prove anything not doubly negated?

As it happens, there is a class of *stable* propositions that are equivalent to their own double negation. Examples include negations, non-strict inequalities on real numbers, and any decidable proposition.

We now see why we required invariant stability in section 2: in the transitivity proof for *cont_trans*, it allows us to employ $\mathit{le\_lt\_dec}_{DN}$ to do case distinction on the t variable when showing that the invariant holds at each point along the composite path. That is, we simply bind $\mathit{le\_lt\_dec}_{DN}\ t\ d$ of type $DN\ (t \leqslant d \vee d < t)$ with the straightforward proof of $(t \leqslant d \vee d < t) \to DN\ (\mathit{invariant}\ (l, \mathit{flow}\ \mathit{system}\ l\ p\ t))$, and then pull the latter out of DN on account of its stability.

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 5 we will see analogous limitations in the choice of one's abstraction parameters.

4 Underestimation and Overestimation

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. This was never a conscious design decision though—it is just the way the development let itself be written. In any case, to characterize tactic-like functions in regular Gallina, we define *underestimation* P to be either a proof of P, or not. In Coq, either of the following will do:

Definition *underestimation* $(P : Prop) : Set := option\ P.$
Definition *underestimation* $(P : Prop) : Set := \{\, b : bool \mid b = true \rightarrow P \,\}.$

The latter tends to work better with the **Program** family of commands [**?**] which have special support for dependent pairs. Using **Program**, an underestimation of the second variety may be provisionally defined strictly as a bool, and then separately proved to be a valid underestimation in a proof obligation generated by **Program**. The second form also nicely illustrates why we call this an underestimation: it may be *false* even when P holds. We can now describe the functionality of our program by saying that it underestimates hybrid system safety, yielding a term of type *underestimation Safe*, where *Safe* is a proposition expressing safety of a hybrid system.

Considered as theorems, underestimations are not very interesting, because they can be trivially "proved" by taking the *false*/*None* estimation. 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 returns *true*/*Just P* for the hybrid system we are interested in (i.e. the thermostat). Again, this is typical tactic stuff. 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 underestimation is an overestimation:

Definition *overestimation* $(P : Prop) : Set := \{\, b : bool \mid b = false \rightarrow \neg P \,\}.$

(Here, too, one could use an option type: *option* $(\neg P)$.)

Since hybrid system safety is defined as unreachability of unsafe states, we may equivalently express the functionality of our development by saying that it overestimates unsafe state reachability. Indeed, most subroutines in our programs will be overestimators rather than underestimators. Notions of overestimation and underestimation trickle down through all layers of our development, down to basic arithmetic. For instance, we employ functions such as:

$$overestimate_{\leqslant_{\mathbb{R}}}\ (\epsilon : \mathbb{Q}^+) : \forall\ x\ y : \mathbb{R}, overestimation\ (x \leqslant_{\mathbb{R}} y)$$

As discussed in the last section, $\leqslant_{\mathbb{R}}$ is not decidable. $overestimate_{\leqslant_{\mathbb{R}}}$ merely makes a "best effort" to prove $\neg(x \leqslant_{\mathbb{R}} y)$ using ϵ-approximations. A smaller ϵ will result in fewer spurious *true* results.

The types of underestimators and overerestimators often merely repeat a certain predicate's parameters, as in the $overestimate_{\leqslant_{\mathbb{R}}}$ example above. Using a bit of type class magic to achieve variadicity, we can rid us of this repetition. Given a predicate $P : A_0 \to ... \to A_n \to Prop$, we define *underestimator* P as the type

$$\forall\ (a_0 : A_0)\ ...\ (a_n : A_n),\ underestimation\ (P\ a_0\ ...\ a_n).$$

(And similar for *overestimator*.) With these, the type of $overestimate_{\leqslant_{\mathbb{R}}}$ may be written as $\mathbb{Q}^+ \to overestimator\ (\leqslant_{\mathbb{R}})$.

We use the same techniques for real decision procedures (in the few places we use those): we define *decision* P as $\{P\} + \{\neg P\}$, and given a predicate $P : A_0 \to ... \to A_n \to Prop$, we define *decider* P as the type $\forall\ (a_0 : A_0)\ ...\ (a_n : A_n)$, *decision* $(P\ a_0\ ...\ a_n)$.

4.1 Automating estimator/decider composition

Underestimators, overestimators and deciders can be combined to form underestimators (resp. overestimators, deciders) for things like conjunctions and quantifications over finite domains. For instance, here is a combinator that forms conjunction overestimators:

Program Definition *overestimate_conj* $\{P\ Q : Prop\}$
  $(x : overestimation\ P)\ (y : overestimation\ Q) :$
    $overestimation\ (P \wedge Q) := x \wedge y.$
Next Obligation.
  intros $[A\ B]$.
  destruct x.*destruct* y.
  simpl **in** H.
  destruct $(andb\_false\_elim\ \_\ \_\ H)$; *intuition*.
Qed.

Applying these combinators by hand to form estimators/deciders for composite propositions is somewhat tedious, but fortunately this process can be automated if we make *overestimation*, *underestimation*, and *decision* type classes:

Class *overestimation* $(P : Prop) : Set$
  $:= overestimate : \{b : bool \mid b = false \to \neg P\}.$

If we now say *overestimate* P, the type class instance resolution mechanism will try to find (or build) a term of type *overestimation* P by recursively applying declared type class instances. For instance, if we declare *overestimate_conj* above as a type class instance (instead of a plain definition) and then say *overestimate* $(A \wedge B)$ where A and B are already known to be overestimatable (meaning the resolution mechanism is able to build terms of types *overestimation* A and *overestimation* B), then the resolution mechanism will be able to build a term of type *overestimation* $(A \wedge B)$ by applying *overestimate_conj*.

We show one nontrivial example to illustrate the convenience of this mechanism: at one point in the development we have Coq automatically construct an overestimator of the property

$$\exists\ u : abstract.State, u \in astates \wedge reachable\ u,$$

simply by saying

$$overestimate\ (\exists\ u : abstract.State, u \in astates \wedge reachable\ u).$$

To construct this overestimator, Coq recursively applies a variety of instances that show that (1) decidable properties are overestimatable; (2) existentially quantified decidable properties over finite domains are decidable; (3) *abstract.State* is a finite domain; (4) conjunctions of decidable properties are decidable; (5) list membership is decidable if equality is decidable for the element type; (6) equality is decidable for products (such as *abstract.State*) if equality is decidable for the components; (7) equality is decidable for *Location*s; (8) equality is decidable for abstract (symbolic) regions; (9) reachability in the abstract system is decidable.

In [3], Asperti et al. use a similar technique (but implemented using unification hints instead of type classes) to automate reflection of algebraic expressions in concrete syntax into a an inductively defined abstract syntax.

5 Abstraction: Regions, States, and Spaces

The abstraction method for verification of (concrete) hybrid system safety as we implement it can be summarized as follows:

1. build an abstract hybrid system corresponding to the concrete hybrid system;
2. show that each trace in the concrete system corresponds to some trace in the abstract system (and as a corollary, that reachability in the concrete system implies reachability in the abstract system, and most importantly, that unreachability in the abstract system implies unreachability in the concrete system);
3. run a certified graph reachability algorithm on the (finite, discrete) abstract system to verify that no unsafe *abstract* states are reachable;
4. conclude from 2 and 3 that no unsafe *concrete* states are reachable either.

An abstract system corresponding to some concrete hybrid system is "like" the concrete system, but with the continuous state space replaced with a finite set of *regions*, each corresponding to a subset of the continuous state space. We begin by describing these regions, after which we will describe abstract transitions and reachability.

5.1 Regions and States

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:

Definition $abstract.State := Location \times Region$.

From now on we will use a "*concrete*." prefix for names like *State* defined in section 2, which now have abstract counterparts. *Region* is a field from a record type *Space* bundling region sets with related requisites:

Record *Space* : *Type* :=
{ *Region* : *Set*
; *Region_eq_dec* : *EqDec Region eq*
; *regions* : *ExhaustiveList Region*
; *NoDup_regions* : *NoDup regions*
; *in_region* : *Container Point Region*
; *in_region_mor* : *Morphism* (*cs_eq* ⟹ *eq* ⟹ *iff*) *in_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 "$x \in y$", prettier than "*in_region x y*". *in_region_mor* states that *in_region* respects *Point*'s setoid equality.

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, that kind of decidability is only available inside *DN* unless all region borders have nontrivial overlap, which as we will see later is undesireable.

Fortunately, the double negation is also *sufficient*, because we will ultimately only use *regions_cover* in a proof of 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*.

We can lift the containment relation between points and regions to a containment relation between concrete states and abstract states:

Instance *abs* : *Container concrete.State abstract.State*
:= $\lambda(l, p)\ (l', r) \Rightarrow l = l' \wedge p \in r$.

Again, the *Container* application reduces to *concrete.State* → *abstract.State* → *Prop*, but the use of *Container* lets us say $s \in s'$ instead of *abs s s'*.

5.2 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;
- if the regions are too coarse, they will fail to capture the subtleties of the hybrid system that actually make it safe (if indeed it is safe at all);
- careless use of region overlap can result in undesireable abstract transitions (and therefore traces), adversely affecting the abstract system's utility (as we will discuss in detail in section 6.2).

Like *Alur*, we use regions formed by multiplying intervals on individual continuous variables. That is, for the thermostat, we first define a *Space* partitioning the continous state space into regions corresponding to clock intervals (which in the development amounts to enumerating the interval bounds, thanks to some generic space definition utilities we wrote), then define a *Space* partitioning the continuous state space into regions corresponding to temperature intervals, and then take the product of these to obtain a *Space* where regions correspond to "squares" in the continuous state space.

In [1], Alur describes a heuristic for interval bound selection, 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 Alur uses, 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. We give one example of why such tweaking was required.

Following the heuristic, Alur derives the following two regions:

$$r_0 := \{(c,t) : \mathit{Point} \mid c \leqslant 0 \wedge 5 \leqslant t \leqslant 6\}$$
$$r_1 := \{(c,t) : \mathit{Point} \mid 2 \leqslant c \leqslant 3 \wedge 6 < t < 9\}$$

In the *Heat* location, where both the clock and the temperature increase linearly, the latter twice as fast as the former, there is a concrete continuous transition from $p_0 := (0,5) \in r_0$ to $p_1 := (2,9)$. p_1 is not an element of r_1, and since the floating point representation Alur uses presumably easily lets one conclude that $5 + 2 * 2 \not< 9$, Alur's procedure manages to determine that there is no continuous transition from any point in r_0 to a point in r_1, justifying suppression of any abstract continuous transition from (Heat, r_0) to (Heat, r_1) (we will discuss abstract transitions in detail in the next section).

Unfortunately, our computable reals only let us compute arbitrarily close approximations of $5+2*2$. Consequently, in the process of determining whether there ought to be an abstract continuous transition from (Heat, r_0) to (Heat, r_1), our procedure (which we will describe in 9) will fail to conclude with certainty that this value does not lie below 9, will thus be unable to definitively establish

the absence of concrete continuous transitions from points in r_0 to points in r_1, and will be forced to include the transition, to ensure that the abstract transition relation respects its concrete counterpart (in a way discussed in detail in the next section).

As it happens, the thermostat's safety indirectly depends on unreachability of (Heat, r_1), and since (Heat, r_0) is a reachable state, the addition of the transition mentioned above makes the abstract system unsafe, obviously preventing us from using it to conclude safety of the concrete system.

To correct this kind of problem, we tweaked many of the bounds, nudging them slightly toward one side or the other. For this particular example, we changed the 9 bound to 8.9. That way, the comparison reduces to $5 + 2 * 2 \not< 8.9$, which a sufficiently close approximation *can* automatically establish.

Another way in which our thermostat regions differ from *Alur*'s lies in the fact that our bounds are always inclusive, which means adjacent regions overlap in lines. We will discuss this in more detail later.

6 Abstract Transitions and Reachability

Once we have a satisfactory abstract *Space*, our goal is to construct an overestimatable notion of abstract reachability implied by concrete reachability, so that concrete unreachability results may be obtained simply by executing the abstract reachability overestimator. It seems reasonable, then, to look for an overestimatable predicate *abstract.reachable* such that *reachable_respect* holds:

Definition *reachable_respect* : *Prop* :=
$\forall\ (s : \mathit{concrete.State}), \mathit{concrete.reachable}\ s \rightarrow$
$\forall\ (s' : \mathit{abstract.State}), s \in s' \rightarrow \mathit{abstract.reachable}\ s'.$

After all, this would imply

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

expressing that to conclude unreachability of a concrete state, one need only establish unreachability of an abstract state that contains it. As we will see shortly, the above definition of *reachable_respect* leads to problems down the line, so this is not actually the definition we use in the development. However, in order to see the motivation for the actual definition we use, let us proceed as if without foresight for a few more moments.

6.1 The Straightforward (but Inadequate) Approach

The definition of *reachable_respect* above suggests that the property we would wish to overestimate is simply

Definition *abstract.reachable* $(s : abstract.State) : Prop$
$:= \exists\ s' : concrete.State, s' \in s \wedge concrete.reachable\ s'.$

In order to overestimate it, we could introduce the following obvious abstract versions of initiality and continuous and discrete transitions:

Definition *abstract.initial* $(s : abstract.State) : Prop$
$:= \exists\ c \in s, concrete.initial\ c.$
Definition *abstract.cont_trans* $: relation\ abstract.State$
$:= \lambda s\ s' \Rightarrow \exists\ (c \in s)\ (c' \in s'), concrete.cont\_trans\ c\ c'.$
Definition *abstract.disc_trans* $: relation\ abstract.State$
$:= \lambda s\ s' \Rightarrow \exists\ (c \in s)\ (c' \in s'), concrete.disc\_trans\ c\ c'.$

We could then prove (by induction over traces) that *abstract.reachable* is included in the transitive closure of $abstract.cont\_trans \cup abstract.disc\_trans$ starting at *abstract.initial*:[3]

Definition *abstract.trans* $: relation\ abstract.State :=$
$abstract.cont\_trans \cup abstract.disc\_trans.$
Definition *reachable_by_abstract_trace* $(s : abstract.State) : Prop$
$:= \exists\ i, abstract.initial\ i \wedge abstract.trans^*\ i\ s.$
Lemma : $abstract.reachable \subseteq reachable\_by\_abstract\_trace.$

This would show that *abstract.reachable* could be overestimated by overestimating *reachable_by_abstract_trace*. For the latter, we would first define an *abstract system* as a triple containing overestimators for *abstract.initial*, *abstract.cont_trans*, and *abstract.disc_trans*:

Record *abstract.System* : *Type* :=
$\{ over\_initial : overestimator\ abstract.initial$
$; over\_cont\_trans : overestimator\ abstract.cont\_trans$
$; over\_disc\_trans : overestimator\ abstract.disc\_trans$
$\}.$

Given an instance of this record, we would then overestimate *reachable_by_abstract_trace* by constructing a graph with vertices representing abstract states and edges representing overestimated abstract transitions, and running an ordinary (but verified) graph reachability algorithm on this graph.

Thus, the task would be reduced to construction of the three overestimators.

6.2 Clouds on the Horizon: Drift

All of the above seems perfectly reasonable, but having gotten to the level of abstract transition overestimators, we can now see where things would (and did,

[3] One would actually take the *alternating* transitive closure, but we will get to that later.

in early versions of our development) go wrong. Unfolding the *overestimator* type specified for the *over_cont_trans* member in *abstract.System*, we get

$$\forall\ (s\ s' : \mathit{abstract.State}), \{\, b : \mathit{bool} \mid b = \mathit{false} \to \neg \mathit{abstract.cont\_trans}\ s\ s' \,\}.$$

Hence, *over_cont_trans* would need to satisfy:

$$\forall\ (s\ s' : \mathit{abstract.State}), \mathit{abstract.cont\_trans}\ s\ s' \to \\ \mathit{abstract.cont\_trans\_over}\ s\ s' = \mathit{true}$$

Which rewrites to:

$$\forall\ (c\ c' : \mathit{concrete.State}), \mathit{concrete.cont\_trans}\ c\ c' \to \\ \forall\ (s\ s' : \mathit{abstract.State}), c \in s) \to c' \in s' \to \\ \mathit{over\_cont\_trans}\ s\ s' = \mathit{true}$$

In other words, the existence of a (continuous) transition from one concrete state to another would force inclusion of transitions between any two abstract states that contain the respective concrete states. When regions do not overlap, this is perfectly appropriate. However, consider the implications in the following example involving overlapping regions.

Suppose our continuous state space is $\mathbb{R}_{\geq 0}$, and our abstract regions are intervals of the form $[n, n+1]$ with $n \in \mathbb{N}$. Further suppose that in some location l, the flow function monotonically increases. Then by the above, *over_cont_trans* must yield *true* given $(l, [1, 2])$ and $(l, [0, 1])$ (in that order), because in l, there actually is a concrete continuous transition from a point in the former to a point in the latter, namely from 1 to 1 (since the continuous transition relation is reflexive). Hence, we would get an abstract transition from $(l, [1, 2])$ to $(l, [0, 1])$, essentially introducing abstract flow in the opposite direction of the concrete flow. This can clearly have disastrous consequences for unreachability of abstract states containing unsafe states, which can render the abstraction useless.

Intuitively, the transition from $(l, [1, 2])$ and $(l, [0, 1])$ seems *redundant*, because the only point in $[0, 1]$ that can be flowed to from $[1, 2]$ is a point that is actually still in $[1, 2]$. Making $[0, 1]$ reachable on account of this redundant transition seems wasteful. Let us formalize this notion of redundance:

Definition $\mathit{redundant\_cont\_trans}\ (l : \mathit{Location}) : \mathit{relation\ Region}$
$:= \lambda r\ r' \Rightarrow \forall\ p, p \in r \to \forall\ t, 0 \leqslant t \to \mathit{flow}\ l\ p\ t \in r' \to \mathit{flow}\ l\ p\ t \in r.$

This relation can be lifted to abstract states. An analogous definition can be given for abstract discrete transitions.

We call the phenomenon of redundant transitions being generated by abstract transition overestimators: "drift". With the specifications for abstract transition overestimators given in the previous section, drift is an inescapable consequence when regions overlap, occurring not only for continuous abstract transitions (as shown above), but also for discrete abstract transitions (as will be discussed in section 10).

It is important to note that drift is not just a consequence of ill-chosen specifications or regions. Suppose we use strictly non-overlapping regions. While

this would solve the problem at the specification level (in that the specification would no longer force the overestimators to produce redundant transitions), the overestimator *implementations* would still be bound to yield redundant transitions when no specific countermeasures are employed. As a trivial example, consider non-overlapping regions [0,1) and [1,2), again with monotonically increasing flow. To justify omission of an abstract continous transition from [1,2) to [0,1), an overestimator implementation would need to prove that $1 \not< 1$, which is not decidable or meaningfully underestimatable given only arbitrarily close approximations of 1.

Hence, our drift countermeasures will be twofold. First, we will first ensure that at the *specification* level, transition overestimators are no longer forced to emit redundant transitions. Then, in sections 9 and 10, we show how our overestimator *implementations* exploit this freedom and use a redundancy hint mechanism to detect and omit redundant transitions.

6.3 Specification Drift Avoidance: Sharing Overestimators

To avoid drift at the specification level, we must either alter the specifications of abstract transition overestimators to account for redundancy (which will necessitate analogous changes in the specifications of abstract reachability and respect, as we will see in a moment), or avoid region overlap.

Practically speaking, using non-overlapping regions implies the use of non-strict equalities, as in the example above. As it happens, non-strict and strict constructive inequalities on real numbers live in different universes: the former live in *Prop*, while the latter live in *Type* . Since derived propositions (such as $p \in r$ for a point p and a region r) all inherit this trait, we would essentially have to give up on *Prop* for the entire development. Unfortunately, several components in the Coq system, including the highly useful subset coercion functionality of the **Program** family of commands, currently only support *Prop*. While these are strictly engineering concerns that may or may not have theoretical substance, they do matter when developing a working system.

On the other hand, changing the abstract transition overestimator specification, abstract reachability, and respect specifications, turns out to be a relatively local change, affecting only a few hundred lines of code, most of them in the one module that introduces these concepts and proves some key lemmas about them.

From an engineering perspective, then, the choice is easy. We will now describe the changes in abstract transition overestimator specification in detail.

In the discussion of the first example in the previous section, we judged the redundant abstract transition from $(l, [1, 2])$ to $(l, [0, 1])$ to be "wasteful" because the destination of the sole concrete continuous transition from 1 to 1 that spawned it was already covered by $[1, 2]$, and so if $[1, 2]$ is reachable, there should be no reason to make $[0, 1]$ reachable as well. Implicit in this intuition is the idea that regions ought to *share* the burden of being reachable on behalf of the points they represent: if 1 is a reachable concrete point, then reachability of

$[1, 2]$ should remove the need for reachability of $[0, 1]$ (to represent reachability of 1), and vice versa.

Unfortunately, as shown in the previous section, the overestimator specifications used in the definition of *abstract.System* given in section 6.1 do not permit this sharing. Let us consider how they would need to be changed to allow for it. We can easily make the unfolded specification of *over_cont_trans* shown in the previous section permit sharing by substituting one universal quantifier with an existential one:

$$\begin{aligned}&\forall\ c\ c', \mathit{concrete.cont\_trans}\ c\ c' \rightarrow\\ &\forall\ (s : \mathit{abstract.State}), c \in s \rightarrow\\ &\exists\ (s' : \mathit{abstract.State}), c' \in s' \wedge\\ &\quad \mathit{over\_cont\_trans}\ s\ s' = \mathit{true}\end{aligned}$$

Applied to the example above, the existence of a concrete continuous transition from 1 to 1 now only results in the requirement that

$$\begin{aligned}&\forall\ (s : \mathit{abstract.State}), (l, 1) \in s \rightarrow\\ &\exists\ (s' : \mathit{abstract.State}), (l, 1) \in s' \wedge\\ &\quad \mathit{over\_cont\_trans}\ s\ s' = \mathit{true}\end{aligned}$$

This means *over_cont_trans* now has the freedom to return *false* given $(l, [1, 2])$ and $(l, [0, 1])$ (thereby suppressing creation of that abstract transition) if it returns *true* given $(l, [0, 1])$ and $(l, [0, 1])$ (which it will have to anyway).

To give *over_cont_trans* and *over_disc_trans* new types that embody this new specification, to replace the old *overestimator* types, we first introduce *shared_cover*:

Definition *shared_cover*
$$\begin{aligned}&\text{`}\{\,\mathit{Container}\ \mathit{concrete.State}\ C\,\}\text{`}\{\,\mathit{Container}\ \mathit{abstract.State}\ D\,\}\\ &\quad (cs : C)\ (ss : D) : \mathit{Prop} :=\\ &\qquad \forall\ s : \mathit{concrete.State}, s \in cs \rightarrow \mathit{DN}\ (\exists\ r : \mathit{abstract.State}, s \in r \wedge r \in ss).\end{aligned}$$

The details of the *Container* type class are of no interest to us right now. What matters is that a "container" of concrete states is said to be sharedly-covered by a "container" of abstract states if for each of the concrete states in the former there is an abstract state in the latter that contains it.[4] We state this in terms of *Container*s so that the definition applies to any types which have a reasonable notion of containment.

Stated in terms of *shared_cover*, a reasonably straightforward type for *over_cont_trans* would be:

$$\begin{aligned}&\mathit{over\_cont\_trans} : \forall\ s : \mathit{abstract.State},\\ &\quad \{\,p : \mathit{abstract.State} \rightarrow \mathit{bool}\ |\\ &\qquad \mathit{shared\_cover}\ (\lambda c' \Rightarrow \exists\ c \in s, \mathit{concrete.cont\_trans}\ c\ c')\ p\,\}\end{aligned}$$

[4] The double negation is here for much the same reasons as were given in section 5.1 for the double negation in the type of *regions_cover*.

That is, *over_cont_trans* would be a function that, given an abstract state, returns a boolean predicate on abstract states that, when interpreted as a container in the obvious way, covers all concrete states reachable by a single concrete continuous transition from a concrete state contained in the original abstract state. We make a few last adjustments to arrive at the actual type used in the development.

First, the abstract transition only actually needs to cover points at which the invariant holds. Integrating that fact in the type of *over_cont_trans* allows us to use *regions_cover* in its implementation:

$$\begin{aligned}&over\_cont\_trans : \forall\ s : abstract.State,\\ &\quad \{\, p : abstract.State \to bool \mid shared\_cover\\ &\qquad (\lambda c' \Rightarrow concrete.invariant\ c' \wedge \exists\ c \in s, concrete.cont\_trans\ c\ c')\ p \,\}\end{aligned}$$

Using some *Container* utilities, we may write this in a more point-free fashion:

$$\begin{aligned}&over\_cont\_trans : \forall\ s : abstract.State,\\ &\quad \{\, p : abstract.State \to bool \mid shared\_cover\\ &\qquad (concrete.invariant \cap (overlap\ s \circ concrete.cont\_trans^{-1}))\ p \,\}\end{aligned}$$

The use of a boolean predicate is less than ideal because it means that to enumerate all abstract states directly reachable from some given abstract state (which we will have to do eventually when computing reachability in a graph whose edges are built from these overestiamtors), one basically needs to filter an exhaustive list of all abstract regions using the predicate, while one could easily imagine that clever overestimator implementations could exploit locality to exclude certain classes of potential transition destinations a priori from consideration. To facilitate such overestimators (which we do not develop in this report), we replace the boolean predicate with a *list* of abstract states. Since lists are also *Container*s, the rest of the definition remains unchanged:

$$\begin{aligned}&over\_cont\_trans : \forall\ s : abstract.State,\\ &\quad \{\, p : list\ abstract.State \mid shared\_cover\\ &\qquad (concrete.invariant \cap (overlap\ s \circ concrete.cont\_trans^{-1}))\ p \,\}\end{aligned}$$

Lists may contain duplicates. The graph reachability algorithm which we will use in section 8 relies on absence of duplicates in the edge relation (expressed as a function returning lists) for its termination proof. While we could filter out duplicates at a much later stage to satisfy that requirement, we choose to let the no-duplicates requirement propagate through to our overestimator specification, avoiding premature pessimization:

$$\begin{aligned}&over\_cont\_trans : \forall\ s : abstract.State,\\ &\quad \{\, p : list\ abstract.State \mid NoDup\ l \wedge shared\_cover\\ &\qquad (concrete.invariant \cap (overlap\ s \circ concrete.cont\_trans^{-1}))\ p \,\}\end{aligned}$$

The *vm_compute* tactic we will ultimately use to run our program tends to reduce terms in *Prop* a bit too eagerly, adversely affecting performance. To work around this, we introduce a bit of laziness in the form of a little unit abstraction:

$$\begin{array}{l} over\_cont\_trans : \forall\ s : abstract.State, \\ \quad \{ p : list\ abstract.State \mid () \to (NoDup\ l \wedge shared\_cover \\ \qquad (concrete.invariant \cap (overlap\ s \circ concrete.cont\_trans^{-1}))\ p) \} \end{array}$$

This will still let us use the property in proofs, but will stop *vm_compute* from unnecessarily evaluating its proof term.

Finally, we actually need the same definition for abstract discrete transitions, so we factor out the common part:

$$\begin{array}{l} \textbf{Definition}\ sharing\_transition\_overestimator \\ \quad (R : relation\ concrete.State) : Set := \forall\ s : abstract.State, \\ \qquad \{ p : list\ abstract.State \mid () \to (NoDup\ l \wedge shared\_cover \\ \qquad\quad (concrete.invariant \cap (overlap\ s \circ R^{-1}))\ p) \}. \end{array}$$

We can now show the definition of *abstract.System* as it appears in the development:

$$\begin{array}{l} \textbf{Record}\ abstract.System : Type := \\ \quad \{ over\_initial : overestimator\ (overlap\ concrete.initial) \\ \quad ; over\_disc\_trans : sharing\_transition\_overestimator\ concrete.disc\_trans \\ \quad ; over\_cont\_trans : sharing\_transition\_overestimator\ concrete.cont\_trans \\ \quad \}. \end{array}$$

6.4 Propagating Sharing Upwards

Consider again the definition of *abstract.reachable* given in section 6.1:

$$\begin{array}{l} \textbf{Definition}\ abstract.reachable\ (s : abstract.State) : Prop \\ \quad := \exists\ s' : concrete.State, s' \in s \wedge concrete.reachable\ s'. \end{array}$$

Clearly, this definition does not let abstract states share the burden of reachability, and can therefore not be overestimated using our retyped abstract transition overestimators. But recall that it followed directly from the following more basic specification we had in mind:

$$\begin{array}{l} \textbf{Definition}\ reachable\_respect : Prop := \\ \quad \forall\ (s : concrete.State), concrete.reachable\ s \to \\ \qquad \forall\ (s' : abstract.State), s \in s' \to abstract.reachable\ s'. \end{array}$$

Which we chose because it implied

$$\begin{array}{l} \forall\ (s : concrete.State) \\ \quad (\exists\ s' : abstract.State, s \in s' \wedge \neg abstract.reachable\ s') \to \\ \qquad \neg concrete.reachable\ s, \end{array}$$

expressing that to conclude unreachability of a concrete state, one need only establish unreachability of *any* abstract state that contains it. However, we now see that this definition, too, neglects to facilitate sharing: when abstract states

may share the burden of reachability, one should establish unreachability of *all* abstract states containing the concrete state. That is, what we really want is an *abstract.reachable* satisfying:

$$\forall\ s : \mathit{concrete.State},\\ (\forall\ s' : \mathit{abstract.State}, s \in s' \to \neg \mathit{abstract.reachable}\ s') \to\\ \neg \mathit{concrete.reachable}\ s.$$

This property follows from the following new definition of *reachable_respect* we will use:

Definition *reachable_respect* : *Prop* :=
shared_cover concrete.reachable abstract.reachable.

This new definition does not give rise to a nice simple definition of *abstract.reachable* distinct from the simple transitive closure of initiality and transition judgements made by the overestimators (analogous to *reachable_by_abstract_trace*), so we just take the latter and prove *reachable_respect* by induction over traces. There is, however, one last subtlety involved.

6.5 Alternating Traces

In the definition of *reachable_by_abstract_trace*, we used the simple transitive closure of the union of *abstract.cont_trans* and *abstract.desc_trans*. This formulation implied that abstract traces could contain successive continuous transitions. Unfortunately, as Alur observes, such repetition results in pathological abstract traces that do not represent any concrete trace.

The solution, as described by Alur, lies in the fact that concrete reachability by arbitrary traces is equivalent to concrete reachability by traces that alternate between continuous and discrete transitions:

Definition *concrete.trans_kind* $(b : \mathit{bool})$: *relation concrete.State* :=
if b **then** *concrete.disc_trans* **else** *concrete.cont_trans*.

Definition *concrete.reachable_alternating* $(s : \mathit{concrete.State})$: *Prop* :=
$\exists\ i : \mathit{concrete.State}, \mathit{concrete.initial}\ i \wedge \mathit{alternate\ concrete.trans\_kind}\ i\ s$.

Lemma *concrete.alternating_reachable_equiv* :
$\forall\ s, \mathit{concrete.reachable}\ s \leftrightarrow \mathit{concrete.reachable\_alternating}\ s$.

(Here, *alternate* forms the alternating transitive reflexive closure of a pair of relations. Indexing the latter by a *bool* simplifies the definition of *alternate*.) As an immediate corollary, we have:

$$\neg \mathit{concrete.reachable\_alternating}\ s \to \neg \mathit{concrete.reachable}\ s$$

Hence, in order to show that unsafe concrete states are not reachable, we need only show that they are not reachable by an alternating trace. Consequently, we may define *abstract.reachable* in terms of alternating traces as well, and still prove *reachable_respect*:

Definition *abstract.trans_kind* $(b : bool) : relation\ abstract.State :=$
$\lambda s\ s' \Rightarrow s \in (\textbf{if}\ b\ \textbf{then}\ over\_cont\_trans\ \textbf{else}\ over\_disc\_trans)\ s'$.

Definition *abstract.reachable* $(s : abstract.State) : Prop :=$
$\exists\ i : abstract.State, over\_initial\ ahs\ i = true \wedge alternate\ abstract.trans\_kind\ i\ s$.

7 Underestimating Safety

In the next section we show that thanks to decidability of our transition and initiality overstimators, *abstract.reachable* is decidable. But first, we end the current section by showing how a decision procedure for *abstract.reachable* lets us underestimate hybrid system safety, and in particular, lets us obtain a proof of thermostat safety. So suppose we have

reachable_dec : *decider abstract.reachable*

And suppose we are given the following specification of unsafe concrete states, covered by a finite list of abstract states:

Variables
(*unsafe* : *concrete.State* $\rightarrow$ *Prop*)
(*astates* : *list abstract.State*)
(*astates_cover_unsafe* : $\forall\ s$, *unsafe* $s \rightarrow \forall\ r, s \in r \rightarrow r \in astates$).

Then, using *reachable_dec* and *unreachable_respect*, we can easily define

Definition *over_unsafe_reachable* : *overestimation* (*overlap unsafe concrete.reachable*).

Taking *unsafe* := *thermo_unsafe* and a suitable abstract cover, we obtain

Definition *over_thermo_unsafe_reachable* :
overestimation (*overlap thermo_unsafe concrete.reachable*).

Recall that *ThermoSafe* was defined as *thermo_unsafe* $\subseteq$ *concrete.unreachable* in section 2. Since we trivially have $\neg$*overlap unsafe concrete.reachable* $\rightarrow$ *ThermoSafe*, we also have:

Definition *under_thermo_unsafe_unreachable* : *underestimation ThermoSafe*.

Finally, we *run* the underestimation:

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

underestimation_true is a tiny utility of type $\forall\ P\ (o : underestimation\ P), o = true \rightarrow P$, whose application in the proof 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 (as we will see in the next section) evaluate the *abstract.System* overestimators (which we will build in sections 9 and 10). 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 overestimators, or plain old 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 overestimators for abstract initiality and transitions bundled in the *abstract.System* record, and the implementation of those overestimators themselves. The former is treated in section 8, the latter in sections 9 and 10.

8 Graphs for Reachability Decision

Given an *abstract.System* containing overestimators for abstract transitions and initiality, we can decide *abstract.reachable* (as defined in the last section) by constructing a directed graph in which vertices and edges correspond to abstract states and transitions, respectively, and then using an ordinary graph reachability algorithm. Because the hybrid system safety proofs we ultimately wish to produce obviously depend on the correctness of the reachability determinations obtained this way, we have implemented a certified graph reachability algorithm inside Coq. For efficiency reasons, it computes a list of all vertices reachable from a given list of initial vertices in one go:

$$\begin{aligned}&\mathit{graph\_reachables} : \forall\ (g : \mathit{DiGraph})\ (\mathit{start} : \mathit{list}\ \mathit{Vertex}), \mathit{NoDup}\ \mathit{start} \to \\ &\quad \{\, l : \mathit{list}\ \mathit{Vertex} \mid \forall\ w : \mathit{Vertex}, w \in l \leftrightarrow \mathit{digraph.reachable}\ \mathit{start}\ w \,\}\end{aligned}$$

Here, *DiGraph* and *digraph.reachable* are defined as

$$\begin{aligned}&\textbf{Record}\ \mathit{DiGraph} : \mathit{Type} := \mathit{Build\_DiGraph} \\ &\quad \{\ \mathit{Vertex} : \mathit{Set} \\ &\quad ;\ \mathit{Vertex\_eq\_dec} : \mathit{EqDec}\ \mathit{Vertex}\ \mathit{eq} \\ &\quad ;\ \mathit{vertices} : \mathit{ExhaustiveList}\ \mathit{Vertex} \\ &\quad ;\ \mathit{edges} : \mathit{Vertex} \to \mathit{list}\ \mathit{Vertex} \\ &\quad ;\ \mathit{edges\_NoDup} : \forall\ v, \mathit{NoDup}\ (\mathit{edges}\ v) \\ &\quad \}.\end{aligned}$$

$$\textbf{Definition}\ \mathit{edge} : \mathit{relation}\ \mathit{Vertex} := \lambda v\ w \Rightarrow w \in \mathit{edges}\ v.$$

$$\begin{aligned}&\textbf{Definition}\ \mathit{digraph.reachable}\ (\mathit{start} : \mathit{list}\ \mathit{Vertex})\ (v : \mathit{Vertex}) : \mathit{Prop} := \\ &\quad \exists\ s \in \mathit{start}, \mathit{edge}^*\ s\ v.\end{aligned}$$

Naively equating vertices with abstract states and edges with abstract transitions (either continuous or discrete) would make vertex reachability in the graph equivalent to potentially-non-alternating abstract state reachability, which as we saw in section 6.5 would result in many needlessly reachable abstract states, potentially rendering the abstraction useless.

To make vertex reachability in the graph equivalent to *alternating* abstract state reachability, we first add a *bool* component to *Vertex*:

Definition $\mathit{Vertex} : \mathit{Set} := \mathit{bool} \times \mathit{abstract.State}$.

Next, we only add edges corresponding to abstract *continuous* transitions from a vertex v with $\mathit{fst}\ v = \mathit{true}$ to a vertex w with $\mathit{fst}\ w = \mathit{false}$, and only add edges corresponding to abstract *discrete* transitions from a vertex v with $\mathit{fst}\ v = \mathit{false}$ to a vertex w with $\mathit{fst}\ w = \mathit{true}$:

Definition $\mathit{nexts}\ (v : \mathit{Vertex}) : \mathit{list}\ \mathit{abstract.State} :=$
  let $(k, s) := v$ **in**
    if k **then** $\mathit{over\_cont\_trans}\ s$
    else $\mathit{over\_disc\_trans}\ s$.

Definition $\mathit{edges}\ (v : \mathit{Vertex}) : \mathit{list}\ \mathit{Vertex} := \mathit{map}\ (\mathit{pair}\ (\mathit{negb}\ (\mathit{fst}\ v)))\ (\mathit{nexts}\ v)$.

Definition $\mathit{graph} : \mathit{DiGraph} := \mathit{Build\_DiGraph}\ \mathit{Vertex}\ \mathit{edges}$.
  (∗ Remaining record fields omitted for the sake of exposition. ∗)

This ensures that paths through the graph alternate between vertices v with $\mathit{fst}\ v = \mathit{true}$ and vertices with $\mathit{fst}\ v = \mathit{false}$, and that the corresponding abstract traces alternate between continuous and discrete transitions. A few simple inductions now show that if we take as *start* those vertices selected by our initiality overestimator, *abstract.reachable* is equivalent to *digraph.reachable*:

Definition $\mathit{start} : \mathit{list}\ \mathit{Vertex} := \{\, v : \mathit{Vertex} \mid \mathit{abstract.over\_initial}\ (\mathit{snd}\ v) \,\}$.

Theorem $\mathit{graph\_respect} : \forall\ s : \mathit{abstract.State}\ \mathit{ap},$
  $\mathit{abstract.reachable}\ s \leftrightarrow \exists\ b : \mathit{bool}, \mathit{digraph.reachable}\ \mathit{start}\ (b, s)$.

Combining *graph_respect*, *abstract.unreachable_respect*, and *reachables*, it is now straightforward to construct a *decider abstract.reachable*.

Through the definitions of *edges*, *nexts*, and *start*, this decider builds on the three overestimators in *abstract.System*. In the next two sections, we discuss how to define an *abstract.System* instance for the discretization (described in section 5.2) of the concrete thermostat's continuous state space.

9 Overestimating Continuous Abstract Transitions

We now discuss the implementation of the second component of an *abstract.System*, *over_cont_trans*:

$\mathit{over\_cont\_trans} : \mathit{sharing\_transition\_overestimator}\ \mathit{concrete.cont\_trans}$

where *sharing_transition_overestimator* is defined as

Definition *sharing_transition_overestimator*
$(R : relation\ concrete.State) : Set := \forall\ s : abstract.State,$
$\{ l : list\ abstract.State \mid () \rightarrow (NoDup\ l \wedge shared\_cover$
$(concrete.invariant \cap (overlap\ s \circ R^{-1}))\ l) \}.$

where *shared_cover* is defined as

Definition *shared_cover*
$`\{ Container\ concrete.State\ C \}`\{ Container\ abstract.State\ D \}$
$(cs : C)\ (ss : D) : Prop :=$
$\forall\ s : concrete.State, s \in cs \rightarrow DN\ (\exists\ r : abstract.State, s \in r \wedge r \in ss).$

9.1 Avoiding Drift

A first observation is that since the concrete continuous transition relation is reflexive, *over_cont_trans* must be reflexive as well (meaning its result list must include the abstract state it received as an argument). By the sharing principle, this means that redundantly reachable regions (whose only reachable points lie within the source region) are accounted for, which in turn means the problem can be reduced to the overestimation of non-redundant transitions, characterized by the following relation:

Definition *substantial_cont_trans* $(l : Location) : relation\ Region$
$:= \lambda r\ r' \Rightarrow \exists\ p\ p', p \in r \wedge p' \in r' \wedge \neg p' \in r \wedge can\_flow\ l\ p\ q.$

Hence, we seek to build an *overestimator substantial_cont_trans*. Note that even though redundantly reachable regions are "accounted for" specification-wise, we still need a mechanism enabling us to notice their redundancy, so that our overestimator may return *false*. Unfortunately, detecting whether a particular abstract continuous transition would be redundant is undecidable in general. For now, let us simply assume existence of a redundance underestimator, and worry about its implementation later:

Variable *under_redundant* : *underestimator redundant_cont_trans*.

The idea now is to build a drift-oblivious *overestimator* for the simplistic *abstract.cont_trans* relation, which *will* emit redundant transitions, but to only run it if the redundancy underestimator does not indicate redundance. That is, supposing we have

Variable *simple_over_cont_trans* : *overestimator abstract.cont_trans*,

we can define

Program Definition *over_subst_cont_trans* $(l : Location)\ (r_{src}\ r_{dst} : Region)$
$: overestimation\ (substantial\_cont\_trans\ l\ r_{src}\ r_{dst})$
$:= \neg under\_redundant\ l\ r\ r' \wedge simple\_over\_cont\_trans\ l\ r_{src}\ r_{dst}.$

Thus, the problem is reduced to construction of *under_redundant* and *simple_over_cont_trans*. We discuss the latter in the next section. For the former, we observe that the redundant transitions we worry about appear as transitions from a region to a region adjacent to it in the opposite direction of the flow corresponding to the current location. This means that if we know which regions are adjacent, and further know when (e.g. in which locations) individual flow components are monotonic (and in which direction), we can underestimate redundancy quite well. Since in our development abstract regions are formed by explicitly enumerating interval bounds, we have a-priori knowledge of which regions will be adjacent. Hence, the only additional work we have to do is to provide as many proofs of flow monotonicity (in the different locations and for the different flow components) as possible, with which some generic utilities can construct a proper redundancy underestimator.

Having taken care of drift, all we need now (to complete the definition of *over_cont_trans*) is an implementation of *simple_over_cont_trans*, which we present in the next section.

9.2 Simple Transition Overestimation

Recall that *abstract.cont_trans* is defined as

Definition $abstract.cont\_trans : relation\ abstract.State$
$:= \lambda s\ s' \Rightarrow \exists\ (c \in s)\ (c' \in s'), concrete.cont\_trans\ c\ c'.$

Since location does not change in continuous transitions, and location equality is decidable (since locations are just names), we can easily reduce the problem down to overestimation of

Definition $region\_cont\_trans\ (l : Location) : relation\ Region$
$:= \lambda r_{src}\ r_{dst} \Rightarrow \exists\ (p_{src} \in r_{src})\ (p_{dst} \in r\_dst'), can\_flow\ l\ p_{src}\ p_{dst},$

where *can_flow* is still

Definition $can\_flow\ (l : Location) : relation\ Point$
$:= \lambda p_{src}\ p_{dst} \Rightarrow \exists\ d : \mathbb{R}_{\geq 0}, flow\ l\ p_{src}\ d = p_{dst} \wedge$
$\forall\ t, 0 \leqslant t \leqslant d \rightarrow invariant\ (l, flow\ l\ p_{src}\ t).$

So given a location l and regions r_{src} and r_{dst}, we are looking to establish absence of concrete transitions from points in r_{src} to points in r_{dst}.

Let us first consider whether we can rule out transitions by looking at the invariant, which *can_flow* demands must hold at every point along the way. Clearly, if we are able to determine that there do not exists paths from points in r_{src} to points in r_{dst} with the invariant holding at each point along the way, then there need be no continuous transition between r_{src} and r_{dst}. Unfortunately, *deciding* this property is too hard. However, suppose we can at least overestimate whether the invariant holds somewhere in an abstract region:

Variable *invariant_overestimator* : *overestimator abstract.invariant*.

With *invariant_overestimator*, we can start defining *simple_over_cont_trans* as follows:

Definition *simple_over_cont_trans* : *overestimator abstract.cont_trans*
:= $\lambda r_{src}\ r_{dst} \Rightarrow$
invariant_overestimator $r_{src} \wedge$
invariant_overestimator $r_{dst} \wedge$ ($*$ .. further tests, to be discussed. $*$).

That is, if either $\neg$*invariant_overestimator* r_{src} or $\neg$*invariant_overestimator* r_{dst}, then we conclude that there can be no concrete transition from a point in r_{src} to a point in r_{dst}. To keep matters simple, we will not bother to check the invariant anywhere else. This does mean that we might generate a spurious abstract transition from r_{src} to r_{dst} if the only reason for lack of concrete transitions between r_{src} and r_{dst} is that the invariant be broken somewhere in the middle of the flow path. Fortunately, this turns out not to cause problems for the thermostat hybrid system. In our development, the definition of *invariant_overestimator* is almost automatic, requiring only that the thermostat's invariant be reformulated in terms of (possibly unbounded) squares in the continuous state space, whose intersection with abstract regions can be automatically overestimated.

Next, let us consider how we might rule out absence of concrete continuous transitions from points in r_{src} to points r_{dst} by looking at the actual flow function. Clearly, if we are able to determine that there are no points in r_{src} which the flow function maps to points in r_{dst}, then there need be no continuous transition between r_{src} and r_{dst}. Equally clearly, this is utterly impossible to meaningfully overestimate for an general flow function and general regions. However, the thermostat's posesses three key properties that we can exploit:

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

A flow function on a product space is *separable* if it can be written as the product of flow function on the respective component spaces. For instance, in $\mathbb{R}^2$, a flow function is separable if future values of the first component only depend on past values of that same component, and the same is true for the second component. Being defined as the product of two flow functions on $\mathbb{R}$ (*temp_flow* and *temp_flow*), the thermostat's flow function (in any location) has this property by construction.

A flow function f on $\mathbb{R}$ is range invertible if

$\exists$ (*range_inverse* : *OpenRange* $\rightarrow$ *OpenRange* $\rightarrow$ *OpenRange*),
$\forall$ (a : *OpenRange*) ($p : \mathbb{R}$), $p \in a \rightarrow$
$\forall$ (b : *OpenRange*) ($d : \mathbb{R}_{\geq 0}$), $f\ p\ d \in b \rightarrow d \in$ *range_inverse* $a\ b$

Here, *OpenRange* represents potentially unbounded intervals in $\mathbb{R}$ (with bounds closed if present). Range invertibility is a less demanding alternative to point invertibility:

$$\exists\,(\mathit{point\_inverse} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}), \forall\ p\ q : \mathbb{R}, f\ p\ (\mathit{point\_inverse}\ p\ q) = q$$

We need range invertibility because the exponential flow functions are not point invertible.

As mentioned in section 2.2, we used 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 overestimation of existence of points in r_{src} which the flow function map to points in r_{dst}. As discussed in section 5.2, the abstract space for our thermostat was constructed as the product of two abstract spaces based on temperature and clock intervals, respectively. Regions in such a product space are pairs of regions in the composite spaces, so r_{src} and r_{dst} can be written as $(r_{src}\_\mathit{temp}, r_{src}\_\mathit{clock})$ and $(r_{dst}\_\mathit{temp}, r_{dst}\_\mathit{clock})$, respectively.

We now simply use an *OpenRange* overlap overestimator of type

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

(defined in terms of things like $\mathit{overestimate}_{\leqslant_{\mathbb{R}}}$ shown in section 4) to overestimate whether the following three ranges overlap:

1. $[0, \infty]$
2. $\mathit{range\_inverse}\ \mathit{temp\_flow}\ r_{src}\_\mathit{temp}\ r_{dst}\_\mathit{temp}$
3. $\mathit{range\_inverse}\ \mathit{clock\_flow}\ r_{src}\_\mathit{clock}\ r_{dst}\_\mathit{clock}$

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* overestimator may justifiably return "false" when the overlap overestimator manages to prove absence of overlap.

Having shown at last the "real" flow test condition used in the overestimation, we can show more precisely how drift would arise if we avoided region overlap (thereby eliminating drift problems at the specification level) but did not bother to employ a redundancy underestimator. Suppose there were clock intervals [0, 1) and [1, 2), and a temperature interval [5, 6). Let us consider what our simple *abstract.cont_trans* overestimator would return given the abstract states $(\mathit{Heat}, (\mathit{half\_range}\ 5\ 6, [1, 2))$ and $(\mathit{Heat}, ([5, 6), [0, 1))$. After determining that the locations are equal as required, and that the invariant holds in both

regions, our overestimator would proceed to overestimate overlap of the ranges mentioned above. In this particular case, with *clock_flow* simply being linear, it would compute that

$$range\_inverse\ clock\_flow\ r_{src}\_clock\ r_{dst}\_clock = [-2, 0)$$

Unfortunately, the overestimator would not be able to determine that [0, inf) does not overlap with [-2, 0), because given only two approximations of 0, it cannot determine that $0 \leqslant 0$. Thus, it would have no choice but to bless the transition.

10 Overestimating Discrete Abstract Transitions

We now turn our attention to the implementation of the *over_disc_trans* component in *abstract.System*. Recall (from section 2.3) the definition of concrete discrete transitions:

Definition *concrete.disc_trans* : *relation concrete.State* := $\lambda s\ s' \Rightarrow$
guard s (*location* s') $\wedge$
reset (*location* s) (*location* s') (*point* s) = *point* s' $\wedge$
invariant $s \wedge$ *invariant* s'.

To overestimate whether there exists a concrete discrete transition from a concrete state in one abstract state to a concrete state in another abstract state, we first have *over_disc_trans* simply check whether the invariant holds at both ends (exactly as in the continuous case), and (in the same way) whether the guard holds at the origin. The interesting part is the overestimation of whether there is a point in the source region that the reset function maps to a point in the destination region. This problem sounds rather similar to that of flow overestimation discussed in section 9, but the similarity is superficial for two reasons: reset functions are much simpler as they don't have a time parameter, and discrete transitions suffer from a substantially different variety of drift.

Suppose one's continuous state space is $\mathbb{R}_{\geq 0}$, and one's abstract regions are intervals of the form $[n, n+1]$ with $n \in \mathbb{N}$. Further suppose that there is a discrete transition from location l to location l' with guard $g = const\ True$ and reset function $r = id$. And suppose we are overestimating whether there needs to be an abstract discrete transition from $(l, [0, 1])$ to $(l', [1, 2])$. Including this transition seems wasteful, because the only point in $[1, 2]$ that the reset function maps a point in $[0, 1]$ to is 1, but that point is still in $[0, 1]$.

For continuous transitions, we avoided drift by explicitly filtering out transitions found to be redundant by a redundance underestimator that relied on reflexivity of the continuous transition relation, knowledge of adjacent regions, and knowledge of where the flow function was monotonic. However, the discrete transition relation is not necessarily reflexive, so we use a slightly different mechanism. Basically, we simply take as an argument an underestimation of whether a particular reset function is the identity function, and if the underestimator

says that it is, then we only make abstract discrete transition to abstract states with the same region as the source state. If on the other hand the underestimator does *not* indicate that the reset function in question is the identity function, then we simply take the bounds of the source region, map them with the reset function, and overestimate whether the resulting region overlaps with the candidate destination (for this to be valid, the reset function must be nondecreasing[5]

For all this to work for a system with a continuous state space in $\mathbb{R}^n$, the reset function must additionally be separable. Our thermostat's reset functions are trivially separable.

11 Conclusions

- nice showcase of proof-by-computation-on-computable-reals
- systematic use of estimators to make tactic-like optional-deciders, at each level in the stack
- computable reals do complicate things
- heuristic for bound selection doesn't work out of the box. manual tweaking obviously not ideal. more experimentation required
- region overlap
- drift and redundance
- code stats(?)
- requirements on system: separability of flow and reset, invariant stability
- lots of room for more clever heuristics with less restrictive preconditions (e.g. separability)

References

1. Alur, R.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embedded Comput. Syst **5** (2006) 2006
2. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.2. (February 2009) http://coq.inria.fr.
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: Hints in unification. In: TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, Berlin, Heidelberg, Springer-Verlag (2009) 84–98
4. Wadler, P.: Monads for functional programming. In Jeuring, J., Meijer, E., eds.: Advanced Functional Programming. Volume 925 of LNCS., Springer (1995) 24–52

[5] When we say that a *flow* function is "nondecreasing", we mean that it does not decrease over time, and so this is really a statement about flow direction. But when we say that a *reset* function (which does not take a time argument) is "nondecreasing", we simply mean that it does not flip ranges around, which is a rather mundane property.