Differential Invariants and Symbolic Integration for Distributed Hybrid Systems

David W. Renshaw **André Platzer**

May 17, 2012
CMU-CS-12-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

This material is based upon work supported by the National Science Foundation by NSF CAREER Award CNS-1054246, NSF EXPEDITION CNS-0926181, and grant nos. CNS-0931985 and CNS-1035800. This research was also supported by the US Department of Transportation's University Transportation Center's TSET grant, award # DTRT12GUTC11. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution or government.

Keywords: Distributed Hybrid Systems, Formal Verification, Theorem Proving

Abstract

We present a formal proof of collision avoidance for a simple distributed hybrid system consisting of an arbitrary finite number of cars on a one dimensional road. Our cars take actions independently from one another and without synchronization, thus behaving in a truly distributed manner. We allow cars to enter and exit the road. For the continuous dynamics, we show how differential invariants and symbolic solutions can be used together harmoniously to prove things that neither could prove alone. We have fully mechanized our formal proof within our theorem prover KeYmaeraD.

1 Introduction

The design of software systems always involves some sort of theorem proving. A system's correctness proof might be muddled and bug-ridden and in fact might only exist in the system designer's mind, but it must exist in some form. Otherwise, the system was merely evolved, not designed. In this light, formal verification is simply the process of making correctness proofs as explicit as possible. We undertake formal verification with the natural expectation that it will be help us to identify and eliminate bugs.

Formalization comes at a cost, of course. It is often claimed that this cost is too high and, more strongly, that formalization turns blatantly true system properties into enormously difficult verification problems. Although we do not argue that formal verification is always called for, we do maintain that it can be quite useful in certain problem domains and that its reputation for injecting difficulty is not entirely deserved.

One domain in which formal verification can be particularly useful is in the design of distributed hybrid systems, where software running on multiple independent agents must interact with a continuously varying environment. In the course of this paper we will walk through the formalization process for a simple example system in this domain. Along the way, we will see that even systems with intuitively straightforward correctness proofs can be very easy to design incorrectly. They can behave in ways that defy our intuition, and formalization helps us catch the problem cases. We will also see that, even though a naïve formalization produces a seemingly intractable verification problem, the properties that are intuitively obvious actually do have proofs that are relatively easy, once they are formulated properly and approached with appropriate methods.

Our walkthrough will feature several new techniques and technologies; the real purpose of the paper is to showcase these. The crux of our approach to the formal proof is a harmonious combination of two methods for dealing with the continuous dynamics: quantified differential invariants [12], and symbolic integration. The former method ensures that intuitively easy facts remain easy to prove, and the latter method gives us a blunt but powerful hammer that allows us to prove the meat of the theorem. We achieved our formalization within the framework of the KeYmaeraD theorem prover [13], adding to it the first mechanized implementation of the method of quantified differential invariants.

2 Designing The System

2.1 Our Task

Suppose we have an infinite straight line, which we will call the *road*, upon which are a finite number of autonomous agents, which we will call *cars*. Each car i has position x_i, velocity v_i, and acceleration a_i. These values implicitly depend on time and evolve according to the system of differential equations

$$\{x_i' = v_i,\, v_i' = a_i,\, a_i' = 0\}. \tag{1}$$

Each car controls itself by adjusting its acceleration. Such adjustments may only occur at certain times—in particular, only once for each *control cycle* of each car. There is a hard upper limit, ε,

on the duration of any given control cycle. This means that when a car makes a control decision it may assume that it will be able to make another control decision within time ε.

We assume that cars have a maximum braking force characterized by a positive constant B, so that at all times we have $a_i \geq -B$ for all cars i. We allow for the possibility of emergency stops; at any time, a car might be forced to brake with acceleration $-B$. We assume moreover that cars have zero length, that their velocity is always nonnegative, and that their control decisions are made with perfect knowledge of the configuration of all other cars.

In this idealized setting, we would like to design car controllers that guarantee the absence of collisions. We would also like to allow for nondeterminism; rather than ask what the *best* choice would be, for some definition of *best*, we ask for a characterization of *all* safe choices. Thus, we ask: at each control decision, what should be the allowed set of accelerations?

2.2 Informal Reasoning

Let us first try to proceed informally. We may formulate our task as follows: given a car f and an acceleration A, we want to know whether A is a safe acceleration for f. Since all cars that are behind f are required to account for the possibility of f making an emergency stop, we do not need to worry about them. Let ℓ be a car that is ahead of f, so that $x_\ell > x_f$. What constraints does ℓ place on A? There are two things that we do not know: 1) the control decisions that ℓ is going to make, and 2) the precise time when f will get to make another control decision. For both of these unknowns, it suffices for us only to consider the worst possible case. Moreover, since the extent of our goal is to prove that f remains safe, it suffices for us to assume that f will make the safest possible choice at its next control decision. Thus, the trajectories of interest can be described as follows: ℓ will immediately begin an emergency stop with acceleration $-B$, and f will continue with acceleration A for the full time ε, after which it will make the safest possible choice, i.e. it will begin an emergency stop with acceleration $-B$. These trajectories continue until both cars come to rest, as illustrated in Figure 1, which shows a particular scenario in which f is indeed able to stop safely before ℓ. We claim that if these trajectories do not cross, then A is a safe acceleration with respect to ℓ.

Note that the trajectories cannot cross twice. For if they do then by the Mean Value Theorem there is a future point in time $\tilde{t}$ when $x_f(\tilde{t}) > x_\ell(\tilde{t})$ and $v_f(\tilde{t}) = v_\ell(\tilde{t})$. Since ℓ has the maximum possible deceleration B, the quantity $x_f - x_\ell$ can never get any smaller after this point, contradicting our assumption that the trajectories cross again. Therefore, we need only check whether the cars f and ℓ are still in the proper order at the *end* of their trajectories, when both cars have come to a stop.

Let us work through the algebra to see what this entails. To do so, we will use the following symbolic solutions to the system's dynamics:

$$\begin{aligned} a_i(t) &= a_i(t_0) \\ v_i(t) &= v_i(t_0) + a_i(t_0)(t - t_0) \\ x_i(t) &= x_i(t_0) + v_i(t_0)(t - t_0) + \tfrac{a_i(t_0)}{2}(t - t_0)^2. \end{aligned} \tag{2}$$

Usually we will suppress the time parameter and assume that we are evaluating at t_0, the point when we are making our decision.

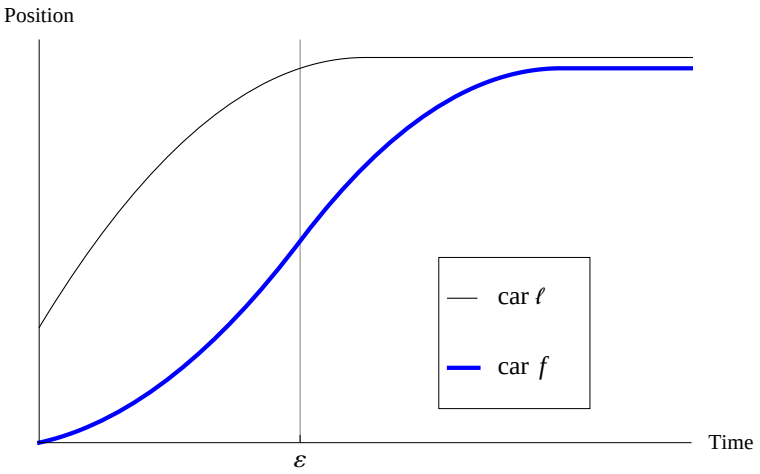

Figure 1: A safe acceleration for f should allow f to stop before hitting ℓ.

The time it takes for ℓ to stop is v_ℓ/B. Therefore, the stopping position of ℓ is

$$x_\ell + v_\ell \frac{v_\ell}{B} + \frac{-B}{2}\left(\frac{v_\ell}{B}\right)^2$$

or, more simply,

$$x_\ell + \frac{v_\ell^2}{2B}. \tag{3}$$

If f continues with acceleration A, then after time ε its position is

$$x_f + v_f\varepsilon + \frac{A}{2}\varepsilon^2$$

and its velocity is

$$v_f + A\varepsilon.$$

Stopping with acceleration $-B$ from a forward velocity $v_f + A\varepsilon$ takes a time of $(v_f + A\varepsilon)/B$ and a distance of $(v_f + A\varepsilon)^2/2B$. This means that the stopping position of f is

$$x_f + v_f\varepsilon + \frac{A}{2}\varepsilon^2 + \frac{(v_f + A\varepsilon)^2}{2B}. \tag{4}$$

Simplifying this expression and combining it with Expression (3) to assert that f stops in front of ℓ gives the following constraint on A:

$$x_\ell + \frac{v_\ell^2}{2B} > x_f + \frac{v_f^2}{2B} + \left(\frac{A}{B} + 1\right)\left(\frac{A\varepsilon^2}{2} + v_f\varepsilon\right). \tag{5}$$

Therefore, if this condition holds for all cars ℓ that are in front of f, then A is a safe acceleration for f. So we are done, right? Wrong! Our reasoning in fact has a subtle and fatal bug. There are some choices for A that satisfy Condition (5) and yet in certain circumstances still allow collisions to occur. We encourage any readers for whom this comes as a surprise to go back and try to find what could go wrong.

3 Quantified Hybrid Programs

3.1 Formalizing the System

We are going to describe our system using the *quantified hybrid program* language defined in [11]. A program in this language describes a system by representing the set of possible *traces* of the system's execution, that is, the set of possible sequences of transitions that the system's state can undergo.

One important kind of transition that our system can undergo is a continuous evolution brought about by its physical dynamics. We might attempt to represent transitions of this kind with the program

$$\mathsf{dynamics}_1 \;\equiv\; \forall i : \mathcal{C}.\{x_i' = v_i,\, v_i' = a_i,\, a_i' = 0\},$$

where we use the symbol $\mathcal{C}$ to represent the set of cars. The transitions represented by $\mathsf{dynamics}_1$ are the evolutions of any duration in which each car i obeys the dynamics we introduced in (1). These are, however, not quite the transitions that we want. Recall that we wish to allow our cars only to move forward. If a_i is negative for some i, then $\mathsf{dynamics}_1$ may transition to a state where some car has negative velocity, violating our system's specification. We need to add a constraint to restrict the set of allowed transitions. We write such constraints to the right of an ampersand, as in the following definition of dynamics, which is what we will actually use in our program:

$$\mathsf{dynamics} \;\equiv\; \forall i : \mathcal{C}\,\{x_i' = v_i,\, v_i' = a_i,\, a_i' = 0,\, s_i' = 1 \,\&\, v_i \geq 0 \wedge s_i \leq \varepsilon\}.$$

Now the evolution can proceed only so long as the constraint $v_i \geq 0 \wedge s_i \leq \varepsilon$ is satisfied. The first part of the constraint, $v_i \geq 0$, ensures that the evolution stops before any car has negative velocity. The second part of the constraint, $s_i \leq \varepsilon$, enforces the upper bound of ε on the time between control decisions; each car i will have a stopwatch s_i tracking the amount of time that has passed since that car's last control decision.

Another kind of transition that our system can undergo is a control flow transition. Two common varieties of such transitions are represented by the branching construct $\cup$, which nondeterministically chooses between two programs, and the looping construct *, which repeats a program a nondeterministic number of times. These constructs are used at the outermost level of our program:

$$\mathsf{cars} \equiv (\mathsf{control} \cup \mathsf{dynamics})^*,$$

Thus, a transition of the cars system consists of any finite sequence of control transitions and dynamics transitions. Of course, because of the constraints we introduced in dynamics, there are some states in which dynamics cannot make a nonnull transition.

All that remains is for us to define the control subprogram. This part of our program demonstrates the three remaining constructs in the hybrid program language: assignment, sequential composition, and conditionals.

$$\mathsf{control} \;\equiv\; i := *_{\mathcal{C}};\quad A := *_{\mathbb{R}};\quad ?\mathsf{Safe}(A, i);\quad a_i := A;\quad s_i := 0.$$

The control program nondeterministically assigns i to a value in $\mathcal{C}$ and A to a value in $\mathbb{R}$. It then tests to see whether A is a safe acceleration for car i, and, if so, assigns a_i to A and restarts

i's stopwatch by assigning s_i to 0, indicating that i has successfully made a control decision. Otherwise, it aborts execution of this branch.

We have reduced our problem to determining an appropriate condition to use for $\mathsf{Safe}(A, i)$. If we give Condition (5) the name $\mathsf{AvoidsAfter}(A, \varepsilon, f, \ell)$, then in the previous section we proposed the following definition:

$$\mathsf{Safe}_1(A, i) \equiv \forall \ell : \mathcal{C} \,.\, x_i < x_\ell \rightarrow \mathsf{AvoidsAfter}(A, \varepsilon, i, \ell).$$

Now that we have posed the problem more formally, can we see what is wrong with this choice?

3.2 Bugs in the Informal Proof

An immediate problem with $\mathsf{Safe}_1(A, i)$ is that it neglects to ensure that B is indeed the maximum possible braking deceleration. Since this is a physical constraint, we kept it implicit in our informal discussion, perhaps misleadingly. In our program, however, we need to be explicit about everything; somewhere we must enforce the fact that $a_i \geq -B$ for all cars i. This is easily accomplished by updating our safety check:

$$\mathsf{Safe}_2(A, i) \equiv A \geq -B \wedge \mathsf{Safe}_1(A, i),$$

because the control branch is the only place in the program where any car's acceleration changes.

A more subtle and insidious problem is that we have, in our informal reasoning, unwittingly required some cars to move in reverse, i.e. with negative velocity. As a result, we have failed to consider some possible trajectories, and some of those that we have considered are not trajectories that the system could actually follow. Figure 2 illustrates a scenario where basing our reasoning on one of these impossible trajectories leads us to make an unsafe decision. The problem arises when car f is able to stop *before* the full control cycle time ε. In this case, if we have $v_f + A\varepsilon < 0$ then Expression (4) does not accurately express f's final position. Instead, it expresses f's final position if f, after braking to a stop, were to continue in reverse with the same negative acceleration A until time ε, and were only then to brake, with positive acceleration B, to a final stop. If car ℓ is stopped between the endpoint of this impossible trajectory and the endpoint of f's actual trajectory, then a collision can occur, as shown in Figure 2.

How did this bug creep into our design? In our informal reasoning we concentrated on dealing with the *worst* case. Perhaps we thought we could always imagine A being positive, because that case seems to be strictly worse than the case when A is negative. However, if A is negative then f's trajectory may be qualitatively different from the typical case, since it might flatten out rather than continue on a quadratic path. That is, after all, what it means to brake to a stop. In that case, the impossible quadratic trajectory might cross ℓ's trajectory twice; if we only look at its endpoint we may mistakenly conclude that A is safe. Note that the *actual* trajectories can still only cross once; *that* part of our reasoning was sound. To fix the bug, we split the safety condition into two cases, depending on whether the acceleration A will cause the car to stop within time ε.

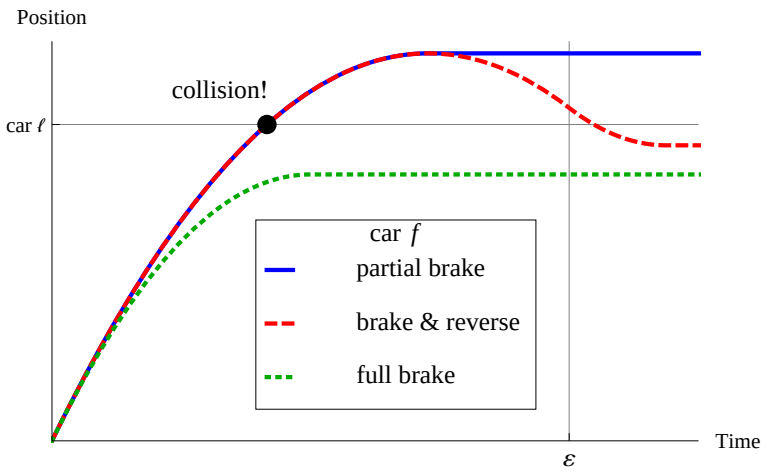

Figure 2: Our informal proof incorrectly considers the endpoint of the impossible trajectory "brake & reverse".

$$\begin{aligned}\mathsf{Safe}(A, i) \;\equiv\; & A \geq -B \wedge \\ & \forall \ell : \mathcal{C}.\; x_i < x_\ell \rightarrow \\ & \qquad (v_i + A\varepsilon \geq 0 \wedge \mathsf{AvoidsAfter}(A, \varepsilon, i, \ell)) \vee \\ & \qquad (v_i + A\varepsilon < 0 \wedge \mathsf{StopsBefore}(A, i, \ell)).\end{aligned}$$

To complete our program cars we now just need to define

$$\mathsf{StopsBefore}(A, i, \ell) \;\equiv\; x_\ell + \frac{v_\ell^2}{2B} > x_i - \frac{v_i^2}{2A},$$

which asserts that the final positions of f and ℓ are in the proper order, assuming that f and ℓ brake until stopping with respective decelerations $-A$ and B.

Are we done now, or are there more bugs? As a sanity check, we might package our assumptions and the assertion that the trajectories do not cross into a formula in first-order real arithmetic and we might pass that formula to a decision procedure, such the one provided by Mathematica. If we do so, the procedure will return after a few minutes with an affirmative answer; those trajectories do not cross. Whew! That puts some of our fears to rest, but we must not forget that the formula only pertains to one specific pair of trajectories. We would like to check our earlier reasoning as well, namely, the reasoning that led us to believe that this pair of trajectories was the only pair we needed to consider. Moreover, working through all of the algebra to determine exactly which formula to send to the decision procedure was a potentially error-prone undertaking. We would like a way to get computerized help for that part of our reasoning, too.

4 Quantified Differential Dynamic Logic

4.1 Formulating Safety

Our goal now will be to formulate and prove a formal theorem stating that cars in our system can never collide with one another. This property can be expressed as a formula in $\mathsf{Qd}\mathcal{L}$, a multimodal logic expressly designed to allow reasoning about traces of quantified hybrid programs [11]. To use $\mathsf{Qd}\mathcal{L}$ for our present purposes, we will only need to introduce one new construct beyond those of standard classical multi-sorted first-order logic. This is the *box* modal operator. For every program α, the modality $[\alpha]$ is defined in terms of the accessibility relation ρ_α given by α's possible executions. If ϕ is any $\mathsf{Qd}\mathcal{L}$ formula, then $[\alpha]\phi$ is true if and only if ϕ is true after any execution of α. That is, if σ is an initial state, then $[\alpha]\phi$ is true at σ if and only if ϕ is true at all states σ' that are accessible from σ by ρ_α. An instance of a box modality appears in our first attempt to write a formula that captures the collision avoidance property:

$$[\mathsf{cars}](\forall i : \mathcal{C}.\ \forall j : \mathcal{C}.\ i \neq j \rightarrow x_i \neq x_j). \tag{6}$$

This formula says that, after any execution of cars, any two distinct cars i, j have distinct positions x_i, x_j. It might seem at first that this formula is inadequate because it only considers the end state, but since the box modality quantifies over all executions and since any legal execution of dynamics may be stopped early to yield another legal execution, the formula does indeed say what we want it to say: that the trajectories of cars do not cross.

To prove a formula is to show that it is *valid*, i.e. true at every state. Unfortunately, Formula (6) is not valid. It fails to include the vital assumptions $B > 0$ and $\varepsilon > 0$ and it does not account for the fact that some initial states simply are not safe. If two distinct cars i, j begin at the same position, for example, then cars i and j can also certainly *end* at the same position, because the null execution is a legal execution of cars. We are therefore forced to make some assumptions about the initial state. We do so by writing the assumptions to the left of an implication arrow $\rightarrow$. The formula that we actually prove will be of the form

$$(B > 0 \wedge \varepsilon > 0 \wedge \mathsf{SafeInit}) \rightarrow [\mathsf{pcars}]\mathsf{NoCollision}, \tag{7}$$

where we still need to define the initial condition SafeInit and revised versions of the program pcars and the collision avoidance condition NoCollision. With SafeInit, we make what is perhaps the simplest possible assumption about the initial state: we assume that the road starts with *no* cars on it. That configuration is trivially safe. We now redefine our program so that it has branches allowing cars to enter and exit the road. Generalizing our system in this way requires some new machinery, but the effort is worthwhile—not only because it vastly simplifies our initial conditions, but also because it gives us tools to reason about multi-lane roads, a topic that interests us anyway. Our new program is

$$\mathsf{pcars} \equiv (\mathsf{exit} \cup \mathsf{enter} \cup \mathsf{control} \cup \mathsf{dynamics})^*.$$

We give each car i a flag p_i indicating whether i is present on the road. The flag p_i can be understood as a boolean field whose value is 1 if car i is present on the road and 0 otherwise. We define

$$\mathsf{SafeInit} \ \equiv \ \forall i : \mathcal{C}.\ p_i = 0$$

so that the system starts with no cars on the road. We must update our postcondition so that it only catches collisions between the cars that are actually present:

$$\mathsf{NoCollision} \;\equiv\; \forall i : \mathcal{C}.\; \forall j : \mathcal{C}.\; i \neq j \wedge p_i = 1 \wedge p_j = 1 \;\rightarrow\; x_i \neq x_j.$$

We now define the new branches. A car that is on the road may exit at any time, but a car may enter the road only if a certain safety condition is satisfied.

$$\mathsf{exit} \;\equiv\; i := *_{\mathcal{C}};\; ?(p_i = 1);\; p_i := 0;\; s_i := 0$$

$$\mathsf{enter} \;\equiv\; i := *_{\mathcal{C}};\; ?(p_i = 0 \wedge \mathsf{SafeToEnter(i)});\; p_i := 1;\; s_i := 0$$

The exit branch chooses an arbitrary car i and checks to see whether i is present. If it is, the branch performs an exit by assigning p_i to 0. We count exits and entrances as control decisions, so the exit branch also restarts the i's stopwatch upon success. The enter branch works in an analogous way but also checks whether $\mathsf{SafeToEnter}(i)$ is true before performing its action. We postpone a formulation of this safety condition until we arrive at a point in the proof where we can better understand what we need from it. During our development of a proof, we will also need to make some minor modifications to the $\mathsf{Safe}(i)$ condition and the SafeInit condition.

4.2 What is a Proof?

Before we begin searching for a proof of Formula (7), we first specify what we mean by *proof*. A proof is a tree of *proof rule* applications whose root contains the formula we wish to prove, our *goal*, and whose leaves contain evidently valid facts. A proof rule is a syntactic procedure for transforming a goal into zero or more subgoals, such that the validity of the subgoals implies the validity of the original goal. The proof rules we use are those of the QdL proof calculus, the details of which are discussed at length elsewhere [11, 13]. The proof calculus does not operate directly on QdL formulas, but rather on *sequents* of the form $\Gamma \Rightarrow \Delta$ where Γ and Δ are finite sets of QdL formulas. Such a sequent denotes the assertion that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ. Proof rules in the calculus generally have the form

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \qquad ... \qquad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta},$$

indicating that to prove $\Gamma \Rightarrow \Delta$, it suffices to prove $\Gamma_i \Rightarrow \Delta_i$ for $i = 1, ..., n$. If we want to prove a formula ϕ, then we may set our initial goal to be the sequent $\cdot \Rightarrow \phi$, where $\cdot$ denotes the empty set.

Our calculus allows us to simplify goals by breaking them into smaller components. For example, one of our proof rules tells us that in order to prove $[\alpha \cup \beta]\phi$ it suffices to prove both $[\alpha]\phi$ and $[\beta]\phi$. During proof search, we attempt to reduce our goal into subgoals that contain none of the special QdL constructs like modalities and indexed variables. If we succeed, we are left with goals that contain only formulas from first-order real arithmetic, which is a decidable theory. In principle, we could include the axioms of real arithmetic in our proof calculus and continue searching for a proof using them. If we were to successfully build a proof in this way, the leaves

of the resulting proof tree would contain only truly obvious facts like $(0{=}1) \Rightarrow \cdot$. That is certainly a desirable property for a proof to have. In practice, however, such an approach is prohibitively expensive because the theory of first-order arithmetic is extremely difficult. Although there is at least one implementation of a proof-producing decision procedure [10], it is orders of magnitude less efficient than optimized solvers such as the one provided by Mathematica. Therefore, we allow subgoals that contain only first-order real arithmetic to be discharged by an external decision procedure. Thus, the sense in which some of the facts at the leaves of our proofs are "evidently valid" is that they have been checked by a trusted decision procedure and could easily be checked again by other independent decision procedures.

5 Towards a Formal Proof

5.1 Building a Loop Invariant

Our first steps toward proving Formula (7) are completely straightforward and require no thought. They consist of a few simple proof rule applications that transform our original goal into the sequent

$$B > 0,\ \varepsilon > 0,\ \mathsf{SafeInit} \ \Rightarrow\ [\mathsf{pcars}]\mathsf{NoCollision}.$$

Here we reach our first nontrivial decision point. To make progress, we must deal with the loop that is at the outermost level of pcars. Doing so requires us to come up with a loop invariant. The proof rule for this situation is

$$\frac{\Gamma \Rightarrow I, \Delta \qquad I \Rightarrow [\alpha]I \qquad I \Rightarrow \phi}{\Gamma \Rightarrow [\alpha^*]\phi,\ \Delta}(^*).$$

This says that in order to prove ϕ is true after the loop α^*, it suffices to find a loop invariant I and to prove that I is initially true, I remains true after a single iteration of the loop, and I implies ϕ. The sets Γ and Δ contain whatever information we know about the initial state, and the comma denotes set insertion, so that ψ, Δ is the set $\{\psi\} \cup \Delta$.

What formulas should we include in our loop invariant? There are a few obvious candidates. In our formalization, we lump the parameters ε and B into the same syntactic category as state variables like x_i and temporary variables like A. Doing so buys us some conceptual simplicity, but also means that we are not syntactically precluded from writing programs like $B := 0$ that assign new values to the parameters. Thus, our program could in principle change these parameters' values during execution. Since our proof will rely heavily on the assumption that ε and B are positive, we must include in the invariant the formula

$$\mathsf{LoopInvA} \ \equiv\ \varepsilon > 0 \ \wedge\ B > 0.$$

Otherwise we would only be able to use this information at the initial state.

Our proof also relies heavily on the fact that each car's state always satisfies some basic sanity conditions. At all times, each car has acceleration greater than or equal to $-B$, velocity greater

than or equal to zero, and stopwatch between zero and ε inclusive. To allow ourselves access to this information later in the proof, we include in our invariant the formula

$$\mathsf{LoopInvB} \;\equiv\; \forall i : \mathcal{C}.\ (a_i \geq -B \wedge v_i \geq 0 \wedge s_i \geq 0 \wedge s_i \leq \varepsilon).$$

Note that this formula requires the sanity conditions to hold for *all* cars, not just those that are present. The idea is that cars not currently on our road still ought to behave like cars. Having this requirement costs us little, and it will turn out to come in quite handy when we are working on the dynamics branch of the proof. The small price we pay is that we must redefine SafeInit to include the facts from LoopInvB.

As yet, our candidate loop invariant does not come close to implying the postcondition NoCollision. Indeed, we have not said anything about how the states of cars compare to each other. We would like somehow to generalize our informal worst-case reasoning about trajectories at control decisions to apply to trajectories at any time. We would like to say that, for any pair of cars, at any time, if the leading car begins an emergency stop, then the following car can slow down in time to avoid it. We can express that property with the formula

$$\begin{aligned}
\mathsf{LoopInvC} \;\equiv\;\; & \forall f : \mathcal{C}.\ \forall \ell : \mathcal{C}. \\
& \quad f \neq \ell \wedge p_f = 1 \wedge p_\ell = 1 \wedge x_f \leq x_\ell \rightarrow \\
& \quad\ x_f < x_\ell \wedge \\
& \quad\ ((v_f + a_f(\varepsilon - s_f) \geq 0 \wedge \mathsf{AvoidsAfter}(a_f, \varepsilon - s_f, f, \ell)) \vee \\
& \quad\ \ (v_f + a_f(\varepsilon - s_f) < 0 \wedge \mathsf{StopsBefore}(a_f, f, \ell))).
\end{aligned}$$

For all pairs of distinct present cars f, ℓ such that f is not ahead of ℓ, the formula LoopInvC tells us that f and ℓ are not colliding at the current moment and they can avoid colliding in the future. It expresses the former property by saying that f is in fact *strictly* behind car ℓ (line 3); this part of the invariant is what will imply our postcondition, i.e. the fact that we have no collisions. It expresses the latter property by saying that even if car ℓ immediately begins braking with deceleration B and car f does not get to make its next control decision until its stopwatch reaches time ε, car f can still avoid a collision (lines 4-5). This part of LoopInvC will be crucial when we need to prove that the invariant remains true after an iteration of the loop. The formula LoopInvC is initially true because initially *no* cars are present.

We now have all of the pieces in place. We define our full loop invariant to be

$$\mathsf{LoopInv} \;\equiv\; \mathsf{LoopInvA} \wedge \mathsf{LoopInvB} \wedge \mathsf{LoopInvC}.$$

With this invariant, we can apply $(^*)$ to yield three new subgoals. The two subgoals

$$B > 0,\ \varepsilon > 0,\ \mathsf{SafeInit} \;\Rightarrow\; \mathsf{LoopInv}$$

and

$$\mathsf{LoopInv} \;\Rightarrow\; \mathsf{NoCollision}$$

are relatively straightforward to prove. The difficult part, and what we will spend most of our subsequent effort on proving, is the inductive subgoal, where we must prove that the invariant remains true after a single iteration of the loop:

$$\mathsf{LoopInv} \Rightarrow [\mathsf{exit} \cup \mathsf{enter} \cup \mathsf{control} \cup \mathsf{dynamics}]\mathsf{LoopInv}.$$

5.2 Using Symbolic Integration

Of the four branches in an iteration of our program's loop, perhaps the most worrisome is the dynamics branch. That is where we expect the most difficult arithmetic to arise, due to the differential equations. Moreover, if we want to stay faithful to our original specification, then our system dynamics are essentially set in stone; unlike in other branches where we may be able to tweak the program to get the proof to work, if our loop invariant fails in the dynamics branch then we are going to need a new loop invariant. Therefore, to catch any potential problems as soon as possible, our next step is to attempt a proof of the sequent

$$\mathsf{LoopInv} \Rightarrow [\mathsf{dynamics}]\mathsf{LoopInv}. \tag{8}$$

We have several proof rules we can apply in this situation. One which seems particularly promising is

$$\frac{\Gamma \Rightarrow S(0), \Delta \qquad \Gamma \Rightarrow \forall t \geq 0.\ [\hat{S}(t)]\big((\forall i{:}\mathcal{C}.H) \rightarrow \phi\big),\ \Delta}{\Gamma \Rightarrow [\forall i : \mathcal{C}\ \{\mathcal{D}\ \&\ H\}]\phi,\ \Delta}(='_{ep}).$$

This rule requires that we provide it a set S of symbolic solutions to the differential equations $\mathcal{D}$, where we assume that the initial time is $t_0 = 0$. To give a concrete example, we note that in our case $S(t)$ will be a conjunction of Equations (2) from page 2 and the stopwatch evolution equation

$$s_i(t) = (t - t_0) + s_i(t_0),$$

all with 0 given for t_0. The $(='_{ep})$ rule has an implicit side condition demanding that the derivative with respect to t of the solutions $S(t)$ must agree with the original differential equations $\mathcal{D}$. This can usually be checked in a straightforward manner by symbolically differentiating S. To ensure that S has the correct constants of integration, the rule's first subgoal checks that $S(0)$ is true in the initial state. Thus, the side condition and the first subgoal together ensure that S is indeed a set of solutions to $\mathcal{D}$.

The second subgoal puts these symbolic solutions to use. Note that we can think of S as a curve running through the state space of the system. We define the program $\hat{S}(t)$ to be an assignment that fast-forwards our state to the point in this curve corresponding to time t. We can imagine $\hat{S}(t)$ turning each equality relation in $S(t)$ into an assignment operator. Now, in order to prove that a formula ϕ is true after the evolution $\forall i : \mathcal{C}\ \{\mathcal{D}\ \&\ H\}$, it certainly suffices for us to prove that ϕ is true after the assignment $\hat{S}(t)$ for all $t \geq 0$. However, doing that proof is not typically possible because nothing prevents the curve S from exiting the region defined by the evolution constraint H, and H usually contains critical information. The $(='_{ep})$ rule takes the constraint into account.

$$
\begin{array}{l}
B > 0,\ \varepsilon > 0,\ s_f \geq 0,\ s_f \leq \varepsilon, \\
a_f \geq -B,\ a_\ell \geq -B,\ v_f \geq 0,\ v_\ell \geq 0, \\
t \geq 0,\ t + s_f \leq \varepsilon,\ v_f + a_f t \geq 0,\ v_\ell + a_\ell t \geq 0, \\
x_f < x_\ell, \\
v_f + a_f(\varepsilon - s_f) \geq 0, \\
x_\ell + \frac{v_\ell^2}{2B} > x_f + \frac{v_f^2}{2B} + \left(\frac{A}{B} + 1\right)\left(\frac{A(\varepsilon - s_f)^2}{2} + v_f(\varepsilon - s_f)\right) \\
\ \Rightarrow \\
\left(x_\ell + v_\ell t + \frac{a_\ell}{2}t^2\right) + \frac{(v_\ell + a_\ell t)^2}{2B} > \\
\left(x_f + v_f t + \frac{a_f}{2}t^2\right) + \frac{(v_f + a_f t)^2}{2B} + \\
\qquad\qquad \left(\frac{A}{B} + 1\right)\left(\frac{A\varepsilon^2}{2} + (v_f + a_f t)\varepsilon\right)
\end{array}
$$

Figure 3: An arithmetic goal that is too difficult for us to handle.

The rule says that in order to prove ϕ is true after the evolution, it suffices to prove the weaker formula $(\forall i{:}\mathcal{C}.H) \rightarrow \phi$ is true after the assignment $\hat{S}(t)$ for all $t \geq 0$. That is, we do not need to consider points that do not satisfy the evolution constraint. We note in passing that rule $(='_{ep})$ is called the *endpoint* rule because we could refine the second subgoal even further to only consider points along S where the curve has not previously exited H; characterizing such points requires us to look at *all* previous points on the curve, rather than just testing whether the endpoint satisfies H. The rule that does this is called the $(=')$ rule.

The $(='_{ep})$ rule led us to a successful proof in our previous work [13]. Therefore it is with high hopes that we apply it to our current goal. Indeed, we may continue building a proof from this point, and for a while everything appears to go well. Using straightforward techniques, we can reduce our goal to very plausible-looking subgoals that consist only of formulas in first-order real arithmetic. Figure 3 shows one of these subgoals. The problem is that these subgoals are too difficult for our solver to handle. Our Mathematica backend fails to return an answer for them, even if it is allowed to run overnight. Unfortunately, there do not seem to be any obvious ways to simplify or further reduce these subgoals before invoking the decision procedure. The situation is perhaps even worse; for all we know, these subgoals might in fact be false, and we might need to invoke the more complicated $(=')$ rule to get the proof to work.

These difficulties have arisen because our loop invariant needs to be more sophisticated than those in previous work. The systems studied in [8] and [13] make use of a globally synchronized control cycle—instead of each car i having a stopwatch s_i, all cars share a global stopwatch s and make their control decisions simultaneously. That unrealistic setup allows the loop invariants to be much simpler. But where does that leave us? Our current system still is quite simple, and we have even come up with an informal correctness proof that is rather convincing. Are we forced to accept the informal proof as the final word on this matter? Especially after we saw how easily we can make mistakes in this domain, that seems hardly acceptable.

6 Quantified Differential Invariants

6.1 Definitions

Recall the style of argument we adopted in our informal proof. We could not hope to consider all possible future events, so we concentrated on what would happen in the worst case. Later, we generalized our worst-case reasoning to derive an important part of our loop invariant. This important part now happens to be what is responsible for generating the difficult arithmetic goals we have just encountered. Does that not seem odd? Should not worst-case reasoning produce invariants that obviously stay true? After all, the worst thing that could happen is precisely the thing we have explicitly considered.

Fortunately, we have another tool at our disposal and it appears to be ideally suited to the current situation. Differential invariants [12] give us a way of using properties that stay true during the evolution of differential equations. We put differential invariants into action with the proof rule

$$\frac{\Gamma \Rightarrow I, \Delta \quad \forall i{:}\mathcal{C}.H \Rightarrow \nabla_{\mathcal{D}} I \quad \Gamma \Rightarrow [\forall i : \mathcal{C}\,\{\mathcal{D}\,\&\,H \wedge I\}]\phi, \Delta}{\Gamma \Rightarrow [\forall i : \mathcal{C}\,\{\mathcal{D}\,\&\,H\}]\phi,\ \Delta}(\nabla).$$

This rule provides another way of proving that ϕ is true after the evolution $\forall i : \mathcal{C}\,\{\mathcal{D}\,\&\,H\}$. Instead of eliminating the evolution, this rule strengthens the evolution's constraint. The first subgoal says that the invariant I must initially be true. The second subgoal says that I must remain true during the evolution of $\mathcal{D}$. This subgoal uses the total derivative operator $\nabla_{\mathcal{D}}$, which we need to define. The third subgoal conjoins the invariant I to the evolution constraint H. Strengthening the constraint is useful because we also have the proof rule

$$\frac{\forall i{:}\mathcal{C}.H \Rightarrow \phi}{\Gamma \Rightarrow [\forall i : \mathcal{C}\,\{\mathcal{D}\,\&\,H\}]\phi,\ \Delta}(\nabla_{close}),$$

which says that if the evolution constraint H implies ϕ then we are done.

Figure 4 gives the definition of the operator $\nabla_{\mathcal{D}}$. Note that $\nabla_{\mathcal{D}}$ transforms a disjunction into a conjunction and an inequation into an equation. On terms, $\nabla_{\mathcal{D}}$ behaves as a standard derivative operator. When it reaches a state variable X, it looks to see whether X appears on the left hand side of an equation in $\mathcal{D}$. If it does not, then $\nabla_{\mathcal{D}}$ returns zero. If it does, then $\nabla_{\mathcal{D}}$ returns the value θ on the right hand side of the equation. When $\nabla_{\mathcal{D}}$ reaches an indexed state variable X_ℓ appearing in an equation $X'_\ell = \theta$ from $\mathcal{D}$, it returns θ with i replaced by ℓ, written θ^ℓ_i. To operate on formulas containing propositional connectives such as $\neg$ and $\rightarrow$ that are not listed in Figure 4, the operator $\nabla_{\mathcal{D}}$ first converts its input into a form that contains only the $\wedge$ and $\vee$ connectives.

6.2 Our Differential Invariants

One nice property of the (∇) rule is that it allows us to add differential invariants in stages. If we have multiple formulas that we believe to be differential invariants, we can first prove that the easier formula is an invariant. Then we will have extra information when we try to prove that more

$$
\begin{array}{rcl}
\nabla_{\mathcal{D}}(F_1 \wedge F_2) & \equiv & \nabla_{\mathcal{D}} F_1 \wedge \nabla_{\mathcal{D}} F_2 \\
\nabla_{\mathcal{D}}(F_1 \vee F_2) & \equiv & \nabla_{\mathcal{D}} F_2 \wedge \nabla_{\mathcal{D}} F_2 \\
\nabla_{\mathcal{D}}(\forall i : \mathcal{C}.\ F) & \equiv & \forall i : \mathcal{C}.\ \nabla_{\mathcal{D}} F \\
\nabla_{\mathcal{D}}(\theta_1 = \theta_2) & \equiv & \nabla_{\mathcal{D}}\theta_1 = \nabla_{\mathcal{D}}\theta_2 \\
\nabla_{\mathcal{D}}(\theta_1 \neq \theta_2) & \equiv & \nabla_{\mathcal{D}}\theta_1 = \nabla_{\mathcal{D}}\theta_2 \\
\nabla_{\mathcal{D}}(\theta_1 < \theta_2) & \equiv & \nabla_{\mathcal{D}}\theta_1 \leq \nabla_{\mathcal{D}}\theta_2 \\
\nabla_{\mathcal{D}}(\theta_1 \leq \theta_2) & \equiv & \nabla_{\mathcal{D}}\theta_1 \leq \nabla_{\mathcal{D}}\theta_2 \\
\nabla_{\mathcal{D}}(\theta_1 + \theta_2) & \equiv & \nabla_{\mathcal{D}}\theta_1 + \nabla_{\mathcal{D}}\theta_2 \\
\nabla_{\mathcal{D}}(\theta_1 \cdot \theta_2) & \equiv & (\nabla_{\mathcal{D}}\theta_1) \cdot \theta_2 \ + \ \theta_1 \cdot (\nabla_{\mathcal{D}}\theta_2) \\
\nabla_{\mathcal{D}} X & \equiv & 0, \quad \text{if } X \notin \mathcal{D} \\
\nabla_{\mathcal{D}} X & \equiv & \theta, \quad \text{if } (X' = \theta) \in \mathcal{D} \\
\nabla_{\mathcal{D}} X_\ell & \equiv & \theta_i^\ell, \quad \text{if } (X_i' = \theta) \in \mathcal{D}
\end{array}
$$

Figure 4: Definitions for $\nabla_{\mathcal{D}}$.

difficult formula is an invariant. In this spirit, to prove Sequent (8) we first apply the (∇) rule with the following formula:

$$\mathsf{EasyDiffInv} \ \equiv \ \varepsilon > 0 \ \wedge \ B > 0 \ \wedge \ \forall i : C.\ (s_i \geq 0 \ \wedge \ a_i \geq -B).$$

This formula, as a consequence of LoopInvA and LoopInvB, is initially true. It remains true during the evolution because its total derivative with respect to dynamics is

$$0 \geq 0 \ \wedge \ 0 \geq 0 \ \wedge \ \forall i : C.\ (1 \geq 0 \ \wedge \ 0 \geq 0),$$

which is obviously valid. Therefore EasyDiffInv is a differential invariant and we may include it in our evolution constraint.

We would now be quite pleased if we could similarly use LoopInvC as a differential invariant. That would eliminate virtually all of our difficulties, as we would then be able to apply (∇_{close}) and be done. Unfortunately, LoopInvC fails to pass muster because of its subformulas $x_f \leq x_\ell$ and $x_f < x_\ell$, whose total derivatives with respect to dynamics do not behave well. Tantalizingly, much of the *rest* of LoopInvC looks ripe for inclusion in a candidate differential invariant. For example, if $\mathcal{D}$ stands for the differential equations from dynamics, then we can compute

$$\nabla_{\mathcal{D}}(\mathsf{AvoidsAfter}(a_f, \varepsilon - s_f, f, \ell)) \quad \rightsquigarrow \quad v_\ell \cdot (B + a_\ell) \geq 0,$$

and this latter formula is clearly valid. How can we take advantage of this observation? We need to add some new data. We introduce a real-valued field n_i for each car i. We think of n_i as the *identification number* of car i, but more importantly we require that it encode order information; we insist at all times that $n_f < n_\ell$ if and only if $x_f < x_\ell$. If cars do not cross trajectories, then

the only time we need to worry about maintaining this ordering is when a car enters the road. We now can go back and redefine LoopInvC to use these identification numbers, replacing the single instance of $x_f \leq x_\ell$ with $n_f \leq n_\ell$. Crucially, we may also define

$$\begin{aligned}\mathsf{DiffInv} \;\equiv\; & \forall f : \mathcal{C}.\ \forall \ell : \mathcal{C}. \\ & f \neq \ell \wedge p_f = 1 \wedge p_\ell = 1 \wedge n_f \leq n_\ell \rightarrow \\ & \quad ((v_f + a_f(\varepsilon - s_f) \geq 0 \wedge \mathsf{AvoidsAfter}(a_f, \varepsilon - s_f, f, \ell)) \vee \\ & \quad\ (v_f + a_f(\varepsilon - s_f) < 0 \wedge \mathsf{StopsBefore}(a_f, f, \ell)))\end{aligned}$$

and use the information from EasyDiffInv to prove that this is a differential invariant. This seems very promising, as DiffInv encodes the worst-case reasoning that led to the difficult arithmetic subgoals responsible for sinking our previous effort. At this point, however, we get stuck. Strengthening our evolution constraint to include both EasyDiffInv and DiffInv does not provide quite enough information for us to successfully close our proof with an application of (∇_{close}). The problem is that we still need to prove that cars present on the road remain in the same order during an evolution. Indeed, this is the meat of the main safety theorem we are trying to establish. More formally, it is the property that any pair of distinct present cars f, ℓ such that $n_f \leq n_\ell$ finishes the evolution with $x_f < x_\ell$. It is hard to see how we could use differential invariants directly to prove this. Thus, although we have found a tidy way to propagate the auxiliary information associated with our worst-case reasoning, we appear once again to have reached an impasse.

6.3 Breakthrough

We have one last trick to employ, and though it perhaps seems straightforward in hindsight, it was surprising to us when we first stumbled upon it. The key is to realize that (∇_{close}) is not the only rule that can help us at this stage in the proof; we can now make progress using the $(='_{ep})$ rule. Since the complicated parts of LoopInvC are now part of the evolution constraint, after applying $(='_{ep})$ we may easily discharge the subgoals that formerly left us with intractably difficult arithmetic. In fact, after applying $(='_{ep})$ the rest of the proof proceeds in a relatively straightforward manner according to methods from previous work [13]. We are done!

Why were we initially blind to this idea? In our prior work, whenever we could write down the solutions to a system's dynamics it usually made sense to immediately use those solutions with a rule like $(='_{ep})$. When using the solutions resulted in intractable arithmetic, or when our system of interest did not have symbolic solutions, we could usually find a differential invariant that implied our desired property. We never used differential invariants and symbolic solutions in combination because we never *needed* to, so the complementarity of their strengths never occurred to us.

$$
\begin{array}{l}
1\quad B>0\ \wedge\ \varepsilon>0\ \wedge\ (\forall i:\mathcal{C}.\ p_i=0\ \wedge\ a_i\geq -B\ \wedge\ v_i\geq 0\ \wedge\ s_i\geq 0\ \wedge\ s_i\leq \varepsilon)\rightarrow\\
2\quad\ [\\
3\quad\ \ (\quad (i:=*_{\mathcal{C}};\ ?(p_i=1);\ p_i:=0;\ s_i:=0)\\
4\quad\ \ \ \cup\\
5\quad\ \ \ (i:=*_{\mathcal{C}};\ N:=*_{\mathbb{R}};\\
6\quad\ \ \ \ ?\Big(p_i=0\ \wedge\ \forall j:\mathcal{C}.\ \Big(j\neq i\ \wedge\ p_j=1\rightarrow\\
7\quad\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \Big(\Big(n_j<N\wedge x_j<x_i\\
8\quad\ \wedge\ \Big(\Big(v_j+a_j\varepsilon\geq 0\ \wedge\ x_i+\frac{v_i^2}{2B}>x_j+\frac{v_j^2}{2B}+\Big(\frac{a_j}{B}+1\Big)\Big(\frac{a_j\varepsilon^2}{2}+v_j\varepsilon\Big)\Big)\ \vee\\
9\quad\ \Big(v_j+a_j\varepsilon<0\ \wedge\ x_i+\frac{v_i^2}{2B}>x_j-\frac{v_j^2}{2a_j}\Big)\Big)\Big)\\
10\quad\ \vee\\
11\quad\ \Big(N<n_j\wedge x_i<x_j\\
12\quad\ \wedge\ \Big(\Big(v_i+a_i\varepsilon\geq 0\ \wedge\ x_j+\frac{v_j^2}{2B}>x_i+\frac{v_i^2}{2B}+\Big(\frac{a_i}{B}+1\Big)\Big(\frac{a_i\varepsilon^2}{2}+v_i\varepsilon\Big)\Big)\ \ \vee\\
13\quad\ \Big(v_i+a_i\varepsilon<0\ \wedge\ x_j+\frac{v_j^2}{2B}>x_i-\frac{v_i^2}{2a_i}\Big)\Big)\Big)\Big)\Big)\Big);\\
14\quad\ \ \ \ p_i:=1;\ n_i:=N;\ s_i:=0)\\
15\quad\ \ \ \cup\\
16\quad\ \ \ (i:=*_{\mathcal{C}};\ A:=*_{\mathbb{R}};\\
17\quad\ \ \ \ ?\Big(A\geq -B\wedge\Big(p_i=1\rightarrow\Big(A=-B\ \vee\\
18\quad\ (v_i=0\ \wedge\ A=0)\ \vee\\
19\quad\ \Big(\forall \ell:\mathcal{C}.\ i\neq\ell\wedge p_\ell=1\wedge n_i\leq n_\ell\rightarrow\\
20\quad\ \Big(\Big(v_i+a_i\varepsilon\geq 0\ \wedge\ x_\ell+\frac{v_\ell^2}{2B}>x_i+\frac{v_i}{2B}+\Big(\frac{A}{B}+1\Big)\Big(\frac{A\varepsilon^2}{2}+v_i\varepsilon\Big)\Big)\ \vee\\
21\quad\ \Big(v_i+a_i\varepsilon<0\ \wedge\ x_\ell+\frac{v_\ell^2}{2B}>x_i-\frac{v_i^2}{2A}\Big)\Big)\Big)\Big)\Big);\\
22\quad\ \ \ \ a_i:=A;\ s_i:=0)\\
23\quad\ \ \ \cup\\
24\quad\ \ \ (\forall i:\mathcal{C}\,\{x_i'=v_i,\ v_i'=a_i,\ a_i'=0,\ s_i'=1\,\&\,v_i\geq 0\wedge s_i\leq\varepsilon\})\\
25\quad\ \ )^*\\
26\quad\ ]\ \forall f:\mathcal{C}.\ \forall \ell:\mathcal{C}.\ f\neq\ell\wedge p_f=1\wedge p_\ell=1\rightarrow x_f\neq x_\ell
\end{array}
$$

Figure 5: The theorem that we proved in KeYmaeraD.

7 The Mechanized Proof

7.1 The Theorem

Figure 5 gives a full statement of the theorem that we formally proved in our theorem prover KeYmaeraD. Before we discuss the mechanization of the proof, we note a few things about the theorem itself. The exit branch of the program (line 3) appears exactly as it did in our exposition. The enter branch (lines 5-14) has a fleshed out $\mathsf{SafeToEnter}(i)$ condition which is slightly more conservative than it strictly needs to be to get the proof to work. Our definition of $\mathsf{SafeToEnter}(i)$ does not require an entering car to have any knowledge of the stopwatch values of any other cars. We feel that, in real systems, such a condition is more implementable than the alternative. The control branch (lines 16-22) has a version of the $\mathsf{Safe}(\mathsf{A},\mathsf{i})$ condition which has been updated to use identification numbers and now has some added options that make liveness of our system more apparent. The dynamics branch (line 24) appears exactly as it did inline.

7.2 The Prover

KeYmaeraD is a theorem prover for QdL [13]. It consists of a small trusted core of data structures, primitive operations, and proof rules. On top of these are non-soundness-critical components, such as a tactic language that facilitates proof search and an interactive GUI. An important feature of KeYmearaD is its ability to work on multiple goals in parallel by utilizing multiple worker subprocesses. KeYmaearaD is written in the Scala programming language.

7.3 Proof Statistics

The mechanized proof is saved as a tactic script comprising 528 lines of Scala code. On an Intel Core2 quad core machine, clocked at 2.83GHz, running Ubuntu Linux 10.04, and using Mathematica 7.0.0 as a backend for first-order real arithmetic, the script takes 195 seconds to execute with 4 workers, and 640 seconds with 1 worker, which indicates good scaling of our parallel proof engine. The completed proof tree has 7154 nodes.

8 Next Steps

8.1 More Complicated Systems

There are many directions in which we could extend our system of cars to make it even more interesting. One idea is to scrap the assumption that there is a universal maximum deceleration B. If different cars have different braking forces, then all of our reasoning about trajectories only being able to cross once becomes invalid. To make a safety proof succeed, we would need to employ more sophisticated reasoning. Another idea is to add multiple lanes to the road. Since each car i already has the field p_i, we could use that to indicate which lane i occupies, as in [8].

8.2 Better Languages and Logics

An annoyance that we often encounter is the need to write the same lengthy formula over and over again. For example, in Figure 5 we have written some form of the AvoidsAfter condition three times. It would be helpful to have abstraction and modularity mechanisms that could eliminate this kind of redundancy. Perhaps we could incorporate into our prover something similar to the definition mechanism that we employ, using the $\equiv$ symbol, throughout our exposition in this paper. Support for that style of reasoning would likely go a long way towards making proofs easier to construct and manipulate.

Another possible area for improvement is in our hybrid program language. A simple variant of our language might distinguish between *parameters* such as B and ϵ, and *state variables* such as x_i and v_i. This would have the benefit of precluding our need to include obvious things like $B > 0$ in invariants. We might then go a step further and refine the class of state variables into *discrete assignables* that only mutate by assignment and *continuous signals* that only mutate during the evolution of differential equations. Future variants of our language might also include better support for describing systems with heterogeneous agents.

9 Related work

Major initiatives have been devoted to developing safe next-generation automated car control systems, including the California PATH project [6], the SAFESPOT and PReVENT initiatives, the CICAS-V system, and many others. With the exception of [8], safety verification for car control systems has been for specific maneuvers or systems with a small number of cars [14, 1, 3, 9]. Our formal verification of collision avoidance applies to a more generic, distributed control for arbitrarily many cars.

Other projects have attempted to ensure the safety of more general systems with simulation and other non-formal methods [4, 6, 2, 7]. Our techniques follow a formal, mechanized, proof calculus, which ensures safety completely, rather than using a finite number of simulations which can only test safety partially.

Wongpiromsarn et al. [15] verify safety of the planner-controller subsystem of a single autonomous ground vehicle. Their verification techniques restrict acceleration changes to fixed and perfect polling frequency, while our model of an arbitrary number of cars allows changes in acceleration at any point in time, with irregular and unsynchronized sensor updates.

Lygeros and Lynch [9] prove safety only for one deceleration strategy for a string of vehicles: the leading vehicle applies maximum deceleration until it stops, while at the same time, the cars following it in the string decelerate to a stop. Dolginova and Lynch [5] verify that no collisions with large relative velocity can occur when two adjacent platoons perform a merge maneuver. This does not prove the absence of small relative velocity collisions, nor anything about the behavior of three platoons or platoons that are not engaged a merge maneuver.

The present paper builds on the work of [8], which gives a cumbersome manual proof of collision avoidance for a highway system, and [13], our subsequent mechanization of a similar proof. The system we consider here is significantly more challenging because it more realistically describes distributed control. Previous work on car collision avoidance verification [9, 15, 8, 13] assumes that all cars reach control decisions at exactly the same points in time. This is an oversimplification that would be impossible to implement in practical distributed systems. In the present paper we study the more general case of an arbitrary number of cars, with appearance and disappearance, where the cars can react at arbitrary independent points in time.

10 Conclusion

After carrying out our proof, we can now confidently say that there are no bugs in the system; our cars are not going to collide. Moreover, we have a powerful framework in place for verifying the design of similar and more sophisticated systems. Crucial to our success has been a practical technique we have developed for using quantified differential invariants in combination with symbolic integration. Key to making approaches for distributed car control implementable has been our reasoning approach for truly distributed control, which always has to allow for cars to perform control decisions at unsynchronized points in time.

Acknowledgments

We thank Sarah Loos for some helpful discussions about her attempts to work with systems similar to the one presented here.

References

[1] ALTHOFF, M., ALTHOFF, D., WOLLHERR, D., AND BUSS, M. Safety verification of autonomous vehicles for coordinated evasive maneuvers. In *IEEE IV'10* (2010), pp. 1078 – 1083.

[2] CHEE, W., AND TOMIZUKA, M. Vehicle lane change maneuver in automated highway systems. PATH Research Report UCB-ITS-PRR-94-22, UC Berkeley, 1994.

[3] DAMM, W., HUNGAR, H., AND OLDEROG, E.-R. Verification of cooperating traffic agents. *International Journal of Control 79*, 5 (May 2006), 395–421.

[4] DAO, T.-S., CLARK, C. M., AND HUISSOON, J. P. Optimized lane assignment using inter-vehicle communication. In *IEEE IV'07* (2007), pp. 1217–1222.

[5] DOLGINOVA, E., AND LYNCH, N. Safety verification for automated platoon maneuvers: A case study. In *HART* (1997), O. Maler, Ed., Springer, pp. 154–170.

[6] HALL, R., AND CHIN, C. Vehicle sorting for platoon formation: Impacts on highway entry and troughput. PATH Research Report UCB-ITS-PRR-2002-07, UC Berkeley, 2002.

[7] JULA, H., KOSMATOPOULOS, E. B., AND IOANNOU, P. A. Collision avoidance analysis for lane changing and merging. PATH Research Report UCB-ITS-PRR-99-13, UC Berkeley, 1999.

[8] LOOS, S. M., PLATZER, A., AND NISTOR, L. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM* (2011), M. Butler and W. Schulte, Eds., LNCS, Springer.

[9] LYGEROS, J., AND LYNCH, N. Strings of vehicles: Modeling safety conditions. In *HSCC* (1998).

[10] MCLAUGHLIN, S., AND HARRISON, J. A proof-producing decision procedure for real arithmetic. In *CADE-20: 20th International Conference on Automated Deduction, proceedings* (Tallinn, Estonia, 2005), R. Nieuwenhuis, Ed., vol. 3632 of *LNCS*, Springer-Verlag, pp. 295–314.

[11] PLATZER, A. Quantified differential dynamic logic for distributed hybrid systems. In *CSL* (2010), A. Dawar and H. Veith, Eds., vol. 6247 of *LNCS*, Springer, pp. 469–483.

[12] PLATZER, A. Quantified differential invariants. In *HSCC* (2011), E. Frazzoli and R. Grosu, Eds., ACM, pp. 63–72.

[13] RENSHAW, D. W., LOOS, S. M., AND PLATZER, A. Distributed theorem proving for distributed hybrid systems. In *ICFEM* (2011), S. Qin and Z. Qiu, Eds., vol. 6991 of *LNCS*, Springer.

[14] STURSBERG, O., FEHNKER, A., HAN, Z., AND KROGH, B. H. Verification of a cruise control system using counterexample-guided search. *Control Engineering Practice* (2004).

[15] WONGPIROMSARN, T., MITRA, S., MURRAY, R. M., AND LAMPERSKI, A. G. Periodically controlled hybrid systems: Verifying a controller for an autonomous vehicle. In *HSCC* (2009), R. Majumdar and P. Tabuada, Eds., vol. 5469, Springer, pp. 396–410.