Distributed Theorem Proving for Distributed Hybrid Systems*

David W. Renshaw, Sarah M. Loos, and André Platzer

Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, USA

Abstract. Distributed hybrid systems present extraordinarily challenging problems for verification. On top of the notorious difficulties associated with distributed systems, they also exhibit continuous dynamics described by quantified differential equations. All serious proofs rely on decision procedures for real arithmetic, which can be extremely expensive. Quantified Differential Dynamic Logic (Qd$\mathcal{L}$) has been identified as a promising approach for getting a handle in this domain. Qd$\mathcal{L}$ has been proved to be complete relative to quantified differential equations. But important questions remain as to how best to translate this theoretical result into practice: how do we succinctly specify a proof search strategy, and how do we control the computational cost? We address the problem of automated theorem proving for distributed hybrid systems. We identify a simple mode of use of Qd$\mathcal{L}$ that cuts down on the enormous number of choices that it otherwise allows during proof search. We have designed a powerful strategy and tactics language for directing proof search. With these techniques, we have implemented a new automated theorem prover called KeYmaeraD. To overcome the high computational complexity of distributed hybrid systems verification, KeYmaeraD uses a distributed proving backend. We have experimentally observed that calls to the real arithmetic decision procedure can effectively be made in parallel. In this paper, we demonstrate these findings through an extended case study where we prove absence of collisions in a distributed car control system with a varying number of arbitrarily many cars.

1 Introduction

Hybrid systems with joint discrete and continuous dynamics have received considerable attention by the research community, including numerous model checking [11, 2, 9] and some theorem proving approaches [18, 22, 23]. Unfortunately, even though hybrid systems verification is already very challenging, not all relevant cyber-physical systems can be modeled as hybrid systems. Hybrid systems cannot represent physical control systems that are distributed or form a multi-agent system, e.g., distributed car control systems. Such systems form *distributed hybrid systems* [8, 15, 25] with discrete, continuous, and distributed dynamics. Distributed hybrid systems combine the challenges of

* This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, Grant Nos. CNS-0926181, CNS-0931985, CNS-1035800, by the ONR award N00014-10-1-0188, by DARPA FA8650-10C-7077. The second author was supported by an NSF Graduate Research Fellowship.

hybrid systems and distributed systems, which are both undecidable. Validation technology for distributed hybrid systems had been mostly limited to simulation [8, 20] and semantic considerations [28, 15]. Very recently, a verification logic, called *quantified differential dynamic logic* (QdL) has been introduced, along with a proof calculus for distributed hybrid systems [25]. This calculus is compositional and has been proved to be complete relative to quantified differential equations [25]. Yet, several questions need to be addressed to translate this theoretical result into practice. We consider questions of automation in a theorem prover in this paper.

The most important question is how to structure and traverse the proof search space for distributed hybrid systems. We develop a range of techniques to control the proof search space in practice. Our first improvement over the QdL base calculus [25] is that we cut down the branching factor during proof search significantly. The QdL base calculus allowed rules to be applied anywhere within a formula, which leads to a substantial amount of unnecessary nondeterminism in proof search. We develop a proper sequent calculus and reduce rule application to top-level formulas in the sequent whenever possible. We dispense with the big step arithmetic rule from [25] and introduce modular arithmetic rules that are more amenable to automation. Instead of recursive first-order substitutions [25], we introduce new proof rules for quantified assignments, which are the distributed and first-order equivalent of Hoare's assignment rule.

These improvements reduce the unnecessary nondeterminism in proof search substantially. Yet, the distributed hybrid systems verification problem also leads to inherent nondeterminisms during proof search. In theory, this concerns only the (in)variant search [25], but, in practice, there are also influential choices in how to handle the arithmetic [24]. The heavy computational cost (doubly exponential) of real arithmetic places quite a burden on the proof search procedure. Especially, common heuristics like "if this branch does not close after 5 min, it (practically) never will" are remarkably unsuccessful in distributed hybrid systems. We need more advanced strategies that consider all proof options in a fair way and timeshare limited computation resources efficiently.

For hybrid systems theorem proving [22], we know several proof strategies that can be successful depending on the property to be shown [24]. We expect different and even more varied proof search strategies to be of relevance in distributed hybrid systems theorem proving. We, thus, develop a strategy language in which new strategies can be expressed easily. In an extended case study, we also show that this strategy language has its merits for scripting local proof tactics for arithmetically difficult parts of a proof.

We take the nondeterminisms in proof search at face value. We develop a proof procedure with built-in and/or-branching. Alternatives in proof rule application produce or-branches. The premises of a particular proof rule produce and-branches. Our approach follows all proof search alternatives in parallel. An alternative will only be discarded if it became irrelevant (an or-sibling has been proved or an and-sibling disproved). Proof search may also temporarily disfavor a proof branch that it considers less promising at the moment but may dynamically revisit this choice later.

We have implemented this approach in a new automated theorem prover called *KeYmaeraD* that has a distributed (multiple cores and computers) proof engine for distributed (multi-agent) hybrid systems. Note that our distributed prover does not just prove one of the distributed agents on each of the distributed cores. This coarse-grained

parallelism is terribly inefficient and not even sufficient, because the systems we consider have an unbounded number of agents, which then could not be proved on a finite computer.

To show that our approach is successful in practice, we consider an extended case study and prove collision freedom in a distributed car control system. Thanks to the distributed proof search procedures in KeYmaeraD, we found a simpler proof than we previously found manually [16]. This observation shows that the approach presented in this paper can be quite useful. Our previous prover, KeYmaera [27], for hybrid systems cannot handle distributed hybrid systems. In previous work on car control verification [16], we, thus, came up with a proof about two cars in KeYmaera and then used a sophisticated modular proof argument showing how safety of the distributed system could be concluded in a modular way. This lifting effort was a formal but fully manual paper-proof and required modularization proof rules that can only be used in some scenarios. In this paper we consider a more systematic approach that makes it possible to verify systems like distributed car control in a fully mechanized theorem prover for distributed hybrid systems, not just hybrid systems. Our contributions are as follows:

- We identify a mode of using $\mathsf{Qd}\mathcal{L}$ proof rules that is suitable for automation and limits the proof search space significantly by reducing unnecessary nondeterminisms.
- We present a systematic proof search framework with and/or-branching that reflects the problem structure in distributed hybrid systems verification naturally.
- We implement our framework in KeYmaeraD, the first verification tool for distributed hybrid systems.
- We present a flexible combinator approach to proof strategies.
- We formally verify collision freedom in a challenging distributed car control system and present the first mechanized proof of distributed car control.

2 Related Work

Hybrid Systems Process-algebraic approaches, like χ [3], have been developed for modeling and simulation. Verification is still limited to small fragments that can be translated directly to other verification tools like PHAVer or UPPAAL, which do not support distributed hybrid systems.

Automated Theorem Proving Theorem provers designed in the so-called *LCF style* focus on the construction of objects of a distinguished type called *thm*, the constructors of which correspond exactly to the proof rules of the logic of interest. This provides an intrinsic mechanism for ensuring that any theorem object represents a valid proof, and it reduces the trusted code base to the implementation of the proof rules. Proof search then centers on the use of *tactics*, which are high-level scripts succinctly describing the expected structure of a proof.

Prominent examples of provers in the LCF style include Isabelle [21] and NuPRL [13]. These systems can be used to encode and reason about object logics such as $\mathsf{Qd}\mathcal{L}$, they permit users to call external decision procedures, and there has been serious work in using parallelism to improve Isabelle's performance [19]. For these reasons, Isabelle

is an attractive candidate for our intended applications. However, the work on parallelism has primarily focused on speeding up the checking of proofs, rather than assisting in the construction of proofs. We would like to use a parallelism model tuned to our particular workflow, and to retain flexibility to modify it in the future. Moreover, we want to move away from the command-line interfaces common to LCF-style provers, instead opting for a more point-and-click interface, akin to that of KeYmaera [27].

Car Control Case Study Major initiatives have been devoted to developing safe next-generation automated car control systems, including the California PATH project, the SAFESPOT and PReVENT initiatives, the CICAS-V system, and many others. With the exception of [16], safety verification for car control systems has been for specific maneuvers or systems with a small number of cars [29, 1, 6, 17]. Our formal verification of collision-freedom applies to a 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 [7, 10, 5, 14]. Our techniques follow a formal, mechanized, proof calculus, which tests safety completely, rather than using a finite number of simulations which can only test safety partially. We build on the work of [16], which presented a cumbersome, manual proof of collision-freedom for a highway system. We generate a semi-automated, mechanized proof safety for a lane of an arbitrary number of cars, where cars may merge into and exit the system. In this case study, mechanization not only provides a more convincing proof, but also allows us to find simpler proofs of safety.

3 Preliminaries: Quantified Differential Dynamic Logic

As a system model for distributed hybrid systems, Qd$\mathcal{L}$ uses *quantified hybrid programs* (QHP) [25]. Note that we use a slightly simplified fragment of Qd$\mathcal{L}$ here that is more amenable to automation. QHPs are defined by the following grammar (α, β are QHPs, θ terms, i a variable of sort C, f is a function symbol, s is a term with sort compatible to f, and H is a formula of first-order logic):

$$\alpha, \beta \ ::= \ \forall i : C\ \mathcal{A} \mid \forall i : C\ \{\mathcal{D}\ \&\ H\} \mid ?H \mid \alpha \cup \beta \mid \alpha;\beta \mid \alpha^*$$

where $\mathcal{A}$ is a list of assignments of the form $f(s) := \theta$ and nondeterministic assignments of the form $f(s) := *$, and $\mathcal{D}$ is a list of differential equations of the form $f(s)' = \theta$. When an assignment list does not depend on the quantified variable i, we may elide the quantification for clarity.

The effect of *assignment* $f(s) := \theta$ is a discrete jump assigning θ to $f(s)$. The effect of *nondeterministic assignment* $f(s) := *$ is a discrete jump assigning *any value* to $f(s)$. The effect of *quantified assignment* $\forall i : C\ \mathcal{A}$ is the simultaneous effect of all assignments in $\mathcal{A}$ for all objects i of sort C. The QHP $\forall i : C\ a(i) := a(i) + 1$, for example, expresses that all cars i of sort C simultaneously increase their acceleration. The effect of *quantified differential equation* $\forall i : C\ \mathcal{D}\,\&\,H$ is a continuous evolution where, for all objects i of sort C, all differential equations in $\mathcal{D}$ hold and formula H holds throughout the evolution (i.e. the state remains in the region described by *evolution domain constraint* H). The dynamics of QHPs changes the interpretation of terms over

time: for an $\mathbb{R}$-valued function symbol f, $f(\boldsymbol{s})'$ denotes the derivative of the interpretation of the term $f(\boldsymbol{s})$ over time during continuous evolution, not the derivative of $f(\boldsymbol{s})$ by its argument $\boldsymbol{s}$. We assume that f does not occur in $\boldsymbol{s}$. In most quantified assignments/differential equations $\boldsymbol{s}$ is just i. For instance, the following QHP expresses that all cars i of sort C drive by $\forall i : C\ x(i)'' = a(i)$ such that their position $x(i)$ changes continuously according to their respective acceleration $a(i)$.

The effect of *test* $?H$ is a *skip* (i.e., no change) if formula H is true in the current state and *abort* (blocking the system run by a failed assertion), otherwise. *Nondeterministic choice* $\alpha \cup \beta$ is for alternatives in the behavior of the distributed hybrid system. In the *sequential composition* $\alpha;\beta$, QHP β starts after α finishes (β never starts if α continues indefinitely). *Nondeterministic repetition* α^* repeats α an arbitrary number of times, possibly zero times.

The formulas of Qd$\mathcal{L}$ [25] are defined as in first-order dynamic logic plus many-sorted first-order logic by the following grammar (ϕ, ψ are formulas, θ_1, θ_2 are terms of the same sort, i is a variable of sort C, and α is a QHP):

$$\phi, \psi \ ::= \ \theta_1 = \theta_2 \mid \theta_1 \geq \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \forall i : C\ \phi \mid \exists i : C\ \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

We use standard abbreviations to define $\leq, >, <, \rightarrow$. The real numbers $\mathbb{R}$ form a distinguished sort, upon which are defined the rigid functions $+$ and $\times$. Sorts $C \neq \mathbb{R}$ have no ordering and hence $\theta_1 = \theta_2$ is the only relation allowed on them. For sort $\mathbb{R}$, we abbreviate $\forall x : \mathbb{R}\ \phi$ by $\forall x\, \phi$. In the following, all formulas and terms have to be well-typed. Qd$\mathcal{L}$ formula $[\alpha]\phi$ expresses that *all states* reachable by QHP α satisfy formula ϕ. Likewise, $\langle\alpha\rangle\phi$ expresses that *there is at least one state* reachable by α for which ϕ holds.

For the formal semantics of Qd$\mathcal{L}$ and QHPs, we refer to [25].

Example 1. Let C be the sort of all cars. By $x(i)$, we denote the position of car i, by $v(i)$ its velocity and by $a(i)$ its acceleration. Then the Qd$\mathcal{L}$ formula

$$(\forall i : C\ \ x(i) \geq 0) \rightarrow [\forall i : C\ \ \{x(i)' = v(i), v(i)' = a(i)\ \&\ v(i) \geq 0\}](\forall i : C\ \ x(i) \geq 0)$$

says that, if all cars start at a point to the right of the origin and we only allow them to evolve as long as all of them have nonnegative velocity, then they end up to the right of the origin. In this case, the QHP just consists of a quantified differential equation expressing that the position $x(i)$ of car i evolves over time according to the velocity $v(i)$, which evolves according to its acceleration $a(i)$. The constraint $v(i) \geq 0$ expresses that the cars never move backwards, which otherwise would happen eventually in the case of braking $a(i) < 0$. This formula is indeed valid, and KeYmaeraD would be able to prove it.

4 Revised Qd$\mathcal{L}$ Proof Calculus

Our desire during verification is to prove that a given formula is valid, that is, true under all interpretations of function symbols. We do this by finding a tree of rule applications (i.e. a proof) within a formal proof calculus (i.e. a set of proof rules), reducing our formula to known facts. In broad strokes, our typical approach is to divide proof search

$$\frac{}{\Gamma, \phi \Rightarrow \phi, \Delta}(close) \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta}(hide\text{-}L) \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \phi, \Delta}(hide\text{-}R)$$

$$\frac{\Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\phi, \Delta}(\neg R) \qquad \frac{\Gamma \Rightarrow \phi, \psi, \Delta}{\Gamma \Rightarrow \phi \vee \psi, \Delta}(\vee R) \qquad \frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \wedge \psi, \Delta}(\wedge R) \qquad \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta}(\rightarrow R)$$

$$\frac{\Gamma \Rightarrow \phi, \Delta}{\Gamma, \neg\phi \Rightarrow \Delta}(\neg L) \qquad \frac{\Gamma, \phi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \vee \psi \Rightarrow \Delta}(\vee L) \qquad \frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta}(\wedge L)$$

$$\frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Rightarrow \Delta}(\rightarrow L) \qquad \frac{x_1 \textbf{ fresh} \qquad \Gamma \Rightarrow \phi(x_1), \Delta}{\Gamma \Rightarrow \forall x : C\ \phi(x), \Delta}(\forall R) \qquad \frac{\Gamma, \forall x : C\ \phi(x), \phi(\theta) \Rightarrow \Delta}{\Gamma, \forall x : C\ \phi(x) \Rightarrow \Delta}(\forall L)$$

$$\frac{[\alpha][\beta]\phi}{[\alpha;\beta]\phi}(\,;\,) \qquad \frac{[\alpha]\phi \wedge [\beta]\phi}{[\alpha \cup \beta]\phi}(\cup) \qquad \frac{\chi \rightarrow \phi}{[?\chi]\phi}(?) \qquad \frac{\Gamma \Rightarrow \psi, \Delta \quad \psi \Rightarrow [\alpha]\psi \quad \psi \Rightarrow \phi}{\Gamma \Rightarrow [\alpha^*]\phi, \Delta}(^*)$$

Fig. 1. Common rules for $\mathsf{Qd}\mathcal{L}$.

into three phases. First we transform and decompose our formula according to any QHPs that it contains. Then we use the nullarize rule (cf. Section 4.6) to get rid of index variables. Finally, we deal with the remaining first-order real arithmetic using quantifier elimination in real-closed fields (which does not support general function symbols [22]).

Taking the proof rules in [25] as a starting point, we have designed new proof rules with several aims in mind. Primarily, we have aimed for a set of proof rules that makes proof search amenable to automation. We have also favored rules that are simple enough that their proof of soundness is readily understood. Pictured in Figure 1 are the proof rules that we leave unmodified (but cf. the caveat in the next subsection), and the standard rules for a classical sequent calculus. Instead of dealing with raw formulas, we deal with *sequents* of the form $\Gamma \Rightarrow \Delta$, denoting that the conjunction of the formulas in the list Γ implies the disjunction of the formulas in Δ, where Γ and Δ are finite sets of formulas. Note that in this paper we concentrate on the $[\alpha]$ modality and universal quantification. Similar ideas apply to the $\langle\alpha\rangle$ modality and existential quantification.

4.1 Working Outside-In

In the proof calculus given in [25], most of the rules for dealing with QHPs can be applied deep within formulas. For example, if we were trying to prove the formula

$$[?x > 30][y := 0 \cup y := x][x := x + 1; ?(y < 10)]\ x = y$$

we could apply the (?) rule, the ($\cup$) rule, or the (;) rule. (In this formula, x and y are nullary functions. For brevity, we do not notationally distinguish between nullary functions and free variables.) In our approach, we only consider the outermost part of a

formula unless we are forced otherwise. So we would use the (?) rule on this formula. This greatly cuts down on the number of choices at each step of proof search. One downside is that sometimes our approach (and-)branches more than is strictly necessary. We find in practice that the benefit from reducing the (or-)branching factor outweighs this cost.

4.2 A Note about Capture

Recall that instead of having a separate syntactic category for state variables, we allow functions to change their interpretation during the execution of a QHP; this is where a program's state is stored. One consequence of this setup is that performing a substitution is not as straightforward as in ordinary first-order logic. We have to worry about functions being captured by assignments inside of modalities. For example, we can incur capture by "substituting" the term $x(i)$ for the variable Y in the formula

$$[\forall j : C\, x(j) := x(j) + 1]\, 0 = Y,$$

even though $x(i)$ does not appear in this formula. If we are not careful, this could lead to unsoundness of our proof rules. Therefore, we use a notion of substitution *admissibility* that excludes substitutions like the above one. We will not formally define admissibility here, but refer to [25].

4.3 Assignment

Proof rules for assignment are central to our approach. We want a proof rule to allow us to work on formulas such as $[x := 1]\phi$. This formula means that ϕ holds after execution of the QHP $x := 1$. One approach to working on this formula would substitute 1 for x in ϕ. Indeed, when doing so is an admissible substitution, this gives us a sound rule. This rule should be familiar to readers familiar with Hoare Logic. If the substitution is not admissible, as in the case when we are trying to prove $[x := 1][x := 0]x = 1$, then this approach fails. In this case, however, we can introduce a new nullary function x_1, rename x to x_1 in ϕ, and instead prove $[x := 1][x_1 := 0]x_1 = 1$, by applying a now-trivial substitution. But then what should we do with formulas such as

$$[x := 1][x := 1 \cup ?(\mathit{true})]x = 1$$

where it is not clear how to rename in a way that will make the substitution admissible? The approach that we take is to delay substitution, encoding its information into a new assumption. Thus, to prove the above formula, we can prove the equivalent

$$(x_1 = 1) \to [x_1 := 1 \cup ?(\mathit{true})]x_1 = 1.$$

We can write our rule as follows:

$$\frac{\mathsf{A}\ \textbf{fresh} \qquad \Gamma, \mathit{updates}(\mathcal{A}, \mathsf{A}) \Rightarrow \mathit{rename}(\mathcal{A}, \mathsf{A}, \phi), \Delta}{\Gamma \Rightarrow [\mathcal{A}]\phi, \Delta}(:=)$$

where A is a set of fresh names for $\mathcal{A}$'s assigned functions. The formula $rename(\mathcal{A}, \mathsf{A}, \phi)$ is ϕ with all occurences of $\mathcal{A}$'s assigned functions renamed by their fresh counterparts (from A). Also, $updates(\mathcal{A}, \mathsf{A})$ is a set of formulas that relates $\mathcal{A}$'s assigned functions to their fresh counterparts in the appropriate way. The exact form $updates(\mathcal{A})$ depends on the form of the assignments contained in it. We show some examples in Figure 2.

$\mathcal{A}$	A	$updates(\mathcal{A}, \mathsf{A})$	$rename(\mathcal{A}, \mathsf{A}, x = f(k))$
$x := x + 1$	x_1	$x_1 = x + 1$	$x_1 = f(k)$
$x := y + 1, y := x$	x_1, y_1	$x_1 = y + 1$ and $y_1 = x$	$x_1 = f(k)$
$\forall i : C\ f(i) := f(i) + 1$	f_1	$\forall i : C\ f_1(i) = f(i) + 1$	$x = f_1(k)$
$f(j) := 3$	f_1	$f_1(j) = 3$ and $\forall i : C\ i \neq j \rightarrow f_1(i) = f(i)$	$x = f_1(k)$

Fig. 2. Examples for the (:=) rule.

4.4 Equality Substitution

The assignment rule ends up adding many new function symbols along with assumptions about them. It is desirable that we have a way to simplify this information. Suppose that θ_1 and θ_2 are closed terms and we know $\theta_1 = \theta_2$. Suppose furthermore that we are trying to prove $\Gamma \Rightarrow \Delta$, where Γ and Δ are modality-free. Then we may replace any occurrence of θ_1 in Γ or Δ with θ_2. Often we want to perform all possible replacements so as to eliminate a particular function. For this common case, we have the following proof rule:

$$\frac{\Gamma^{\theta_2}_{\theta_1} \Rightarrow \Delta^{\theta_2}_{\theta_1}}{\theta_1 = \theta_2, \Gamma \Rightarrow \Delta}(=)$$

Here, $\Gamma^{\theta_2}_{\theta_1}$ means Γ with every occurrence of θ_1 replaced by θ_2. This is not a substitution in the ordinary sense of the word, because θ_1 is a term, not a variable. We emphasize that it is important that Γ and Δ be modality-free. Otherwise the rule could incur capture and be unsound.

4.5 Differential Equations

Suppose that the QHP we need to deal with is a set of quantified differential equations $\mathcal{D}$, and suppose furthermore that $\mathcal{D}$ has a set of symbolic solutions $S(t)$. The usual proof rule to apply in this situation, as put forth in [25], is

$$\frac{\forall t \geq 0\ ((\forall 0 \leq \tilde{t} \leq t\ [S(\tilde{t})]H) \rightarrow [S(t)]\phi)}{[\forall i : C\ \{\mathcal{D}\ \&\ H\}]\phi}(=')$$

which is essentially a direct translation of the semantics of $\mathcal{D}$. The premise can be understood informally as follows: for all future times t, if the solution remains in the

domain constraint H up to time t, then the postcondition is true at time t. This premise has the undesirable characteristic of containing a nested quantification on the left of an implication. Often, the following rule (with a simpler, but stronger premise) suffices:

$$\frac{\forall t \geq 0\ [S(t)](H \rightarrow \phi)}{[\forall i : C\ \{\mathcal{D}\ \&\ H\}]\phi}(='_{endpoint})$$

This premise states that, for all future times t, if the solution is in the domain constraint H at t, then the postcondition is true at t. We call this the *endpoint* version of the rule.

4.6 Eliminating Index Variables

The first order theory of real numbers is decidable only for formulas that have no uninterpreted non-nullary function symbols. Therefore, in order to use a backend decision procedure, we need to get rid of such functions. In [25], this task was accomplished in a proof rule that eliminated all non-nullary functions in a single proof rule application. This had the potential to cause an exponential blowup in the size of the sequent.

In contrast, we take a more local approach. We use what we call the *nullarize* proof rule, which looks for occurrences of a given closed term θ, and replaces them with a new nullary function. We write the rule as follows.

$$\frac{g_1\ \textbf{fresh} \qquad \Gamma^{g_1}_{\theta} \Rightarrow \Delta^{g_1}_{\theta}}{\Gamma \Rightarrow \Delta}(null)$$

Recall that this is not substitution—it is a replacement operation. It is important that θ be a closed term. We may not, for example, use the rule to get rid of $f(i)$ in the formula

$$\forall i : C\ \ f(i) > 0,$$

If this formula occurred on the left of the sequent, then we can nullarize f only after we have used the $(\forall L)$ rule to instantiate i.

4.7 Real Arithmetic

Nullary functions can be understood as being implicitly universally quantified. In contrast, we consider any *variables* that are free to be implicitly existentially quantified (inside of the universal quantification of functions). For example, if Y is a free variable and x is a nullary function, then the formula $x = Y$ means for all interpretations of x there exists a value for Y such that $x = Y$. This particular formula is valid.

Thus, once we have eliminated modalities and non-nullary functions, we are left with a sequent that is equivalent to a formula in the first-order theory of real closed fields. This is a decidable theory. Note that there is a subtle distinction here—first-order logic over the reals, with uninterpreted functions, is undecidable. However, first-order arithmetic, with only the rigid arithmetic functions, is decidable. Therefore, when we have reached this point, we invoke a decision procedure for this theory.

5 Proving in KeYmaeraD

In order to make use of the above proof calculus, we have implemented KeYmaeraD, a new theorem prover. KeYmaeraD's design is inspired by the LCF approach to theorem proving. At any given time there is a tree called the *proof state*, which the user is trying to build into a proper proof. Each node in the tree represents a proof goal (i.e. a sequent). The only way the user has of changing the proof state is to apply one of the proof rules, as pictured in Figure 3. Applying a proof rule to a goal does one of three things:

1. fails, in which case the proof state is left unchanged,
2. succeeds in closing the goal,
3. breaks the goal into one or more conjunctive subgoals.

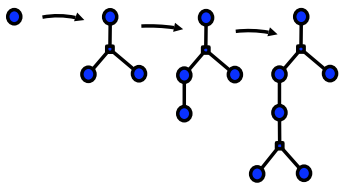

Fig. 3. Application of proof rules to nodes (the circles) leads to and-branching (the squares).

One way in which KeYmaeraD differs from many LCF-style provers is that it allows or-branching on the proof state itself, rather than only at the level of tactics. (We will discuss tactics later.) This allows the user to explore multiple possible proofs simultaneously, as pictured in Figure 4. If any or-branch successfully closes, KeYmaeraD automatically marks the others as irrelevant, as pictured in Figure 5.

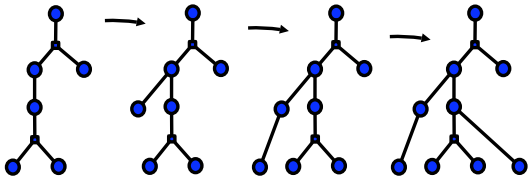

Fig. 4. Application of different rules at the same node leads to or-branching, shown here as circle nodes with two children.

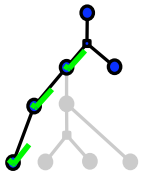

Fig. 5. Closing an or-branch.

The typical strategy we use to try to prove a QdL formula in KeYmaeraD is as follows: We want to use the proof rules to get rid of modalities and the indices. Then we will be left with arithmetic, which is decidable. For dealing with the QHPs in the modalities, we have found that it often suffices to work from the outside in, as discussed

in the previous section. In this way, we do not have to think about which rule to apply. The hard part in this phase is choosing invariants for loops and differential equations (if they do not have symbolic solutions; see [26]). The or-branching is useful for trying different invariants and remembering why particular branches fail to close.

Next, we get rid of indices. The hard part here is choosing instantiations. If our goal has no modalities and no indices, we can pass it to the arithmetic backend. This procedure will return asynchronously and KeYmaeraD will appropriately update the proof state to reflect its success or failure.

One key observation is that giving the arithmetic solver *too much* information can cause it to take too long (even by > 3 orders of magnitude). We often need to decide what parts of the sequent to include. A common pattern in our workflow is the following. We let the arithmetic decision procedure work on a sequent as soon as we have gotten rid of modalities and indices. In the meantime, we start an or-branch that hides believed-to-be-irrelevant formulas in the sequent before again invoking the procedure. Sometimes the original call returns before we even get to making a second call. Sometimes the second call returns immediately and makes the first call irrelevant.

6 Strategy Language

Using KeYmaeraD to apply proof rules one by one—a task that is already much easier than manipulating QdL formulas on paper—quickly becomes tedious. To increase the user's power, we introduce tactics, which are a way to script proof search. Our ultimate aim is to allow QdL theorem proving to be as automated as possible. We envision that future versions of KeYmaeraD will be able to perform successful proof searches that take on the order of days or weeks using a cluster of tens or hundreds of computers running the arithmetic backend. Carefully designed tactics should provide a modular way to work toward this aim.

KeYmaeraD has an embedded language of base tactics and tactic-combinators ("tacticals" in the jargon). Tactics can be built from provided tactics. We also allow tactics to use arbitrary code in the Scala language, which is the implementation language of KeYmaeraD. In this latter case, tactics can read and do whatever analysis they like on the entire proof state. In keeping with our LCF-style design, the only way that a tactic can change the proof state, however, is to apply proof rules. Hence, tactics are not soundness-critical, only important for completeness.

The type of a tactic is as follows:

$$Node \to Option[List[Node]]$$

A tactic takes a node of the proof state. It does some computation that might have effects on the proof state. Then, it either returns `None`, signaling failure, or it succeeds and returns a list of nodes. Note that this success does not necessarily signify that the tactic has proved any particular goal—it just means that the tactic did what it was meant to do on this node. The list of returned nodes is intended to be used for the composition of tactics. A typical mode of use is for this list to be a conjunctive list of subgoals— if they are all valid then the original sequent is valid. However, the soundness of the system does not depend on tactics being used only in this way. Indeed, the tactic `arithT`,

as explained below, does not follow this pattern. Some common tactics and tactic combinators are shown in Figure 6.

Tactics	
`nilT`	Always fails.
`unitT`	Always succeeds, returning the given node (no-op).
`tryruleatT(rl,pos)`	Tries to apply the rule `rl` at the position `pos`.
`tryruleT(rl)`	Tries to apply the rule `rl` at all positions until it succeeds.
`tryrulepT(rl,prd)`	Tries the rule on a top-level formula where the predicate is true.
Tactic Combinators	
`eitherT(t1,t2)`	First tries `t1`. Upon failure, tries `t2`.
`composeT(t1,t2)`	Tries `t1` and then, upon success, applies `t2` to all the returned nodes.
`repeatT(t)`	Tries `t` until it fails, returning the result of the final success.
`branchT(t,ts)`	Tries `t`. Upon success, maps the returned nodes to `ts`.

Fig. 6. Tactics and Tactic Combinators

6.1 Example: Instantiation

Here we explain one kind of tactic that we have found useful. Suppose we have several formulas with universal quantifiers on the left in our sequent. To make use of these formulas, we will need to use the ($\forall L$) rule, which instantiates the formulas. We may want to think carefully about what terms we will use for the instantiations. If so, we might, e.g. use a tactic that looks for formulas of the form $i = j$ uses these matched terms to instantiation. At the other extreme, we may just want to instantiate the quantified formulas with any and all terms that could possibly make sense. (This is often feasible for sorts other than the real numbers, where no functions are predefined and the only relation is equality.) This is often useful in sequents that are light on arithmetic. But exhaustive instantiation quickly chokes the arithmetic solvers. Tactics that take either of these tacks are used heavily in our case study.

6.2 Arithmetic

The decision procedure for arithmetic returns asynchronously. We have a tactic called `arithT` that fails if the goal cannot be passed to the procedure. (This happens if there are any modalities or indices left.) Otherwise it succeeds, returning the empty list. When the procedure returns with a result, KeYmaeraD will automatically update the proof state to reflect the new information. To better understand this protocol, consider the composed tactic `eitherT(arithT, myOtherTactic)`. If `arithT` fails, then we need to continue to work on the sequent to get rid of modalities or indices. Therefore, in that case we continue with `myOtherTactic`. Otherwise, we do not need to do anything other than wait for the decision procedure to return. So the tactic succeeds, even if the procedure eventually returns "false."

6.3 Input Formulas

Some important proof rules such as (*) are parametrized by a formula. Because of our renaming method for dealing with assignments, sometimes it is impossible for the writer of a proof script to know in advance what name some functions will have when such a rule needs to be applied. Therefore, we provide a unification function `unify` that can be invoked in tactics. The result of `unify(fm1,fm2)` is either failure or a substitution function which, when applied to `fm2` will return `fm1`. Note that we do not use `unify` to synthesize invariant formulas, we merely use it to get a handle on formulas as they shift names.

7 Case Study

In this section we present a mechanized, formal verification of a distributed adaptive cruise control and automatic braking system as a complex case study of the KeYmaeraD theorem prover. Major initiatives have been devoted to developing safe next-generation automated car control systems, including the California PATH project, the SAFESPOT and PReVENT initiatives, the CICAS-V system, and many others. Chang et al. [4], for instance, propose CICAS-V in response to a report that crashes at intersections in the US cost $97 Billion in the year 2000.

Providing a formal verification of safety-critical cyber-physical systems is vital to ensure safety as the public adopts these systems into daily use. However, before KeYmaeraD, formal verification of large-scale, distributed, hybrid systems was only possible manually. Manual proofs not only require ample skilled man-power, but are also prone to errors. Applying the powerful verification methods of QdL to a broad range of distributed hybrid systems is not possible without automation and mechanization, which KeYmaeraD provides.

In this section, we present the first semi-automated and fully mechanized proof of an arbitrary number of cars driving under distributed controllers along a straight lane.

Modeling Model 1 is a QHP for an arbitrary number of cars following distributed, discrete and continuous dynamics along a straight lane. In addition, the model allows cars to appear and disappear at any time and in any safe location, simulating lane changes. The discrete control consists of three possible choices, modeled as a nondeterministic assignment in $ctrl(i)$; see line (3). Braking is allowed at all times, and is the only option if certain safety constraints are not met. Car i may accelerate only if the constraint $\mathbf{Safe}_\varepsilon(i)$ holds, meaning that the cars in front of car i are far enough away for car i to accelerate for at least ε time units. Here, ε represents the upper bound on sensor/communication update delay. Additionally, if the car is stopped, it can always continue to stand still.

Every car on the lane is associated with three real values: position, velocity, and acceleration. Since cars may also appear and disappear, we add a fourth element: existence. The existence field is a bit that flips on ($\mathsf{E}(n) := 1$) when a car appears on the lane and off ($\mathsf{E}(n) := 0$) when a car disappears. Any number of cars may disappear from the road (simulating merging into an adjacent lane or exiting the highway) at any time.

Model 1 Local highway control (`lhc`)

$$\texttt{lhc} \equiv ((ctrl^n; dyn^n)^* \cup delete^* \cup create^*)^* \tag{1}$$

$$ctrl^n \equiv \forall i : C\ \ a(i) := *;\ ?(\forall i : C\ \ \mathsf{E}(i) = 1 \rightarrow ctrl(i)) \tag{2}$$

$$ctrl(i) \equiv a(i) = -B\ \vee\ (\mathbf{Safe}_\varepsilon(i) \wedge a(i) = A)\ \vee\ (v(i) = 0 \wedge a(i) = 0) \tag{3}$$

$$\mathbf{Safe}_\varepsilon(i) \equiv \forall j : C\ \ x(i) \leq x(j) \wedge i \neq j \rightarrow \tag{4}$$

$$x(i) + \frac{v(i)^2}{2B} + \left(\frac{A}{B} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon v(i)\right) < x(j) + \frac{v(j)^2}{2B} \tag{5}$$

$$dyn^n \equiv (t := 0;\ \forall i : C\ \ \{dyn(i)\}) \tag{6}$$

$$dyn(i) \equiv x(i)' = v(i),\ v(i)' = a(i), t' = 1\ \&\ (t \leq \varepsilon\ \wedge\ (\mathsf{E}(i) = 1 \rightarrow v(i) \geq 0)) \tag{7}$$

$$delete \equiv n := *;\ ?(\mathsf{E}(n) = 1);\ \mathsf{E}(n) := 0 \tag{8}$$

$$create \equiv n := new;\ ?(\forall i : C\ \ E(i) = 1 \rightarrow (i \ll n) \wedge (n \ll i)) \tag{9}$$

$$(n := new) \equiv n := *;\ ?(\mathsf{E}(n) = 0 \wedge v(n) \geq 0);\ \mathsf{E}(n) := 1 \tag{10}$$

$$(i \ll n) \equiv (x(i) \leq x(n) \wedge i \neq n) \rightarrow \left(x(i) < x(n) \wedge x(i) + \frac{v(i)^2}{2B} < x(n) + \frac{v(n)^2}{2B}\right) \tag{11}$$

To accomplish this, the model non-deterministically chooses an existing car and flips its existence bit to off; see line (8). Modeling cars merging into the lane is almost as simple; however, before a car can merge, it must check that it will be safely in front of or behind all previously existing cars on the lane; see line (9).

This model is similar to the `lhc` model proved manually in [16], but with a few simplifications. First, we assume in line (4) that the cars have omniscient sensing, i.e., each car receives data about the position and velocity of all the cars on the lane, as opposed to just the car directly ahead. Second, we assume that when the car accelerates, it applies maximum acceleration, and when it brakes it applies maximum braking, rather than choosing from a bounded range of acceleration and braking forces.

Verification Now that we have described a suitable model for a lane of cars in a highway (Model 1), we identify a set of safety requirements and prove that the model never violates them. Safety verification must ensure that, at all times, every car on the road is safely behind all the cars ahead of it in its lane. We say that car i is safely following car j if $(i \ll j)$, as defined in line (11). To capture the notion that the cars should be safe at all times, we use the $[\alpha]$ modality, as shown in Proposition 1.

Proposition 1 (Safety of local highway control `lhc`). *Assuming the cars start in a controllable state (i.e. each car is a safe distance from the cars ahead of it on the lane), the cars may move, appear, and disappear as described in the (`lhc`) model, then no cars will ever collide. This is expressed by the following provable* QdL *formula:*

$$(\forall\ i : C\ \forall j : C\ (\mathsf{E}(i) = 1 \wedge \mathsf{E}(j) = 1) \rightarrow ((i \ll j) \wedge v(i) \geq 0 \wedge v(j) \geq 0))\ \rightarrow$$
$$[\texttt{lhc}](\forall i : C\ \forall j : C\ (\mathsf{E}(i) = 1 \wedge \mathsf{E}(j) = 1) \rightarrow ((i \ll j) \wedge v(i) \geq 0 \wedge v(j) \geq 0)))$$

Our final tactic script is about 400 lines. At the end of its execution, the proof state has 1134 nodes. On a MacBook Pro with a 2.86GHz Core 2 Duo processor, using Math-

ematica 7.0.0 for the real arithmetic backend, the proof takes 40 seconds to complete with one worker, and 33 seconds with two workers. This includes the time it takes to compile and load the tactic script— approximately 13 seconds.

In the course of developing this proof, we discovered that the endpoint rule for differential equations suffices for this formula—a simplification which greatly increases the computational efficiency of our proof.

Because KeYmaeraD uses a tactics-based approach rather than real-time interactions, verification requires fewer human inputs and lends itself to reusability. The two car case for this model, for instance, required far fewer tactics when implemented in KeYmaeraD than the hundreds of human-interactions needed by KeYmaera. The tactics were also robust enough to be applied to multiple proof branches. Moreover, we initially proved a version that omitted $x(i) < x(j)$ in the invariant. Then, after realizing that the invariant did not obviously imply the safety condition we wanted, we added this condition. With only minimal changes to the tactics script, the updated model was easily verified.

The manual proof presented in [16] relies heavily on modular proof structure principles to get the proof complexity to a manageable size. With KeYmaeraD, we can improve on that modular structure by employing modular proof tactics. This approach still simplifies the resulting proof structure as before, but, unlike dedicated modularity arguments, it also maintains better robustness to changes in the model.

8 Conclusions and Future Work

We introduce automation techniques for theorem proving for distributed hybrid systems using quantified differential dynamic logic. We have implemented KeYmaeraD, the first formal verification tool for distributed hybrid systems. As a major case study in KeYmaeraD, we have formally verified collision freedom in a sophisticated distributed car control system with an unbounded (and varying) number of cars driving on a straight lane.

References

1. Althoff, M., Althoff, D., Wollherr, D., Buss, M.: Safety verification of autonomous vehicles for coordinated evasive maneuvers. In: IEEE IV'10. pp. 1078 – 1083 (2010)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138(1), 3–34 (1995)
3. van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schiffelers, R.R.H.: Syntax and consistent equation semantics of hybrid Chi. J. Log. Algebr. Program. 68(1-2), 129–210 (2006)
4. Chang, J., Cohen, D., Blincoe, L., Subramanian, R., Lombardo, L.: CICAS-V research on comprehensive costs of intersection crashes. Tech. Rep. 07-0016, NHTSA (2007)
5. Chee, W., Tomizuka, M.: Vehicle lane change maneuver in automated highway systems. PATH Research Report UCB-ITS-PRR-94-22, UC Berkeley (1994)
6. Damm, W., Hungar, H., Olderog, E.R.: Verification of cooperating traffic agents. International Journal of Control 79(5), 395–421 (May 2006)

7. Dao, T.S., Clark, C.M., Huissoon, J.P.: Optimized lane assignment using inter-vehicle communication. In: IEEE IV'07. pp. 1217–1222 (2007)
8. Deshpande, A., Göllü, A., Varaiya, P.: SHIFT: A formalism and a programming language for dynamic networks of hybrid automata. In: Hybrid Systems. pp. 113–133 (1996)
9. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT 10(3), 263–279 (2008)
10. Hall, R., Chin, C.: Vehicle sorting for platoon formation: Impacts on highway entry and troughput. PATH Research Report UCB-ITS-PRR-2002-07, UC Berkeley (2002)
11. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. In: LICS. pp. 394–406 (1992)
12. Hespanha, J.P., Tiwari, A. (eds.): Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings, vol. 3927. Springer (2006)
13. Howe, D.J.: Automating Reasoning in an Implementation of Constructive Type Theory. Ph.D. thesis, Cornell University (1988)
14. Jula, H., Kosmatopoulos, E.B., Ioannou, P.A.: Collision avoidance analysis for lane changing and merging. PATH Research Report UCB-ITS-PRR-99-13, UC Berkeley (1999)
15. Kratz, F., Sokolsky, O., Pappas, G.J., Lee, I.: R-Charon, a modeling language for reconfigurable hybrid systems. In: HSCC. pp. 392–406 (2006)
16. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) FM. LNCS, Springer (2011)
17. Lygeros, J., Lynch, N.: Strings of vehicles: Modeling safety conditions. In: HSCC (1998)
18. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: HSCC. pp. 305–318 (1998)
19. Matthews, D.C.J., Wenzel, M.: Efficient parallel programming in Poly/ML and Isabelle/ML. In: DAMP (2010)
20. Meseguer, J., Sharykin, R.: Specification and analysis of distributed object-based stochastic hybrid systems. In: HSCC. pp. 460–475 (2006)
21. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5 (1989)
22. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. 41(2), 143–189 (2008)
23. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. 20(1), 309–352 (2010)
24. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
25. Platzer, A.: Quantified differential dynamic logic for distributed hybrid systems. In: Dawar, A., Veith, H. (eds.) CSL. LNCS, vol. 6247, pp. 469–483. Springer (2010)
26. Platzer, A.: Quantified differential invariants. In: Frazzoli, E., Grosu, R. (eds.) HSCC. ACM (2011)
27. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer (2008)
28. Rounds, W.C.: A spatial logic for the hybrid π-calculus. In: HSCC. pp. 508–522 (2004)
29. Stursberg, O., Fehnker, A., Han, Z., Krogh, B.H.: Verification of a cruise control system using counterexample-guided search. Control Engineering Practice (2004)

9 Erratum (January 20, 2012)

In the published version of this paper, the (*) proof rule in Figure 1 incorrectly includes Γ and Δ in all three premises. That version of the rule is unsound. KeYmaeraD implements the correct version of the rule, so the results of the paper are unaffected.