Differential Refinement Logic *

Sarah M. Loos
Computer Science Department
Carnegie Mellon University
sloos@cs.cmu.edu

André Platzer
Computer Science Department
Carnegie Mellon University
aplatzer@cs.cmu.edu

Abstract

We introduce *differential refinement logic* (dRL), a logic with first-class support for refinement relations on hybrid systems, and a proof calculus for verifying such relations. dRL simultaneously solves several seemingly different challenges common in theorem proving for hybrid systems: 1. When hybrid systems are complicated, it is useful to prove properties about simpler and related subsystems before tackling the system as a whole. 2. Some models of hybrid systems can be implementation-specific. Verification can be aided by abstracting the system down to the core components necessary for safety, but only if the relations between the abstraction and the original system can be guaranteed. 3. One approach to taming the complexities of hybrid systems is to start with a simplified version of the system and iteratively expand it. However, this approach can be costly, since every iteration has to be proved safe from scratch, unless refinement relations can be leveraged in the proof. 4. When proofs become large, it is difficult to maintain a modular or comprehensible proof structure. By using a refinement relation to arrange proofs hierarchically according to the structure of natural subsystems, we can increase the readability and modularity of the resulting proof. dRL extends an existing specification and verification language for hybrid systems (differential dynamic logic, dL) by adding a refinement relation to directly compare hybrid systems. This paper gives a syntax, semantics, and proof calculus for dRL. We demonstrate its usefulness with examples where using refinement results in easier and better-structured proofs.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

1. Introduction

As new technologies become indispensable in the next generation of safety-critical cyber-physical applications such as collision avoidance maneuvers for aircraft and emergency braking or adaptive cruise control in the automotive industry, it is of increasing importance that these systems be guaranteed error-free. To ensure there are no mistakes in the models on which these systems are based (most commonly hybrid systems models [1]), formal verification methods like model checking and theorem proving have been developed to prove systems safe under a continuously infinite range of circumstances. To verify safety for cyber-physical systems, there is an existing specification and verification logic for hybrid systems: differential dynamic logic (dL) [19, 21, 22]. The logic dL supports behavioral reasoning about hybrid programs α with modalities, where the formula $[\alpha]\phi$ expresses that all states after running α satisfy formula ϕ. Previously, dL and its proof calculus have had many successes in proving safety for sophisticated cyber-physical systems, such as distributed car and aircraft control. However, verification results for these complicated systems often rely on equally complicated and creative proofs. Such proofs need considerable foresight to ensure that proofs about simpler systems magically fit to just the right shape required for simplifying subsequent arguments about more complex or reoccurring systems. *However, with direct proof support for relating two systems, such implicit structure in the proofs could be formalized explicitly, thereby simplifying the proofs significantly.*

In this paper we introduce *differential refinement logic* (dRL), which extends dL with a *refinement operator* on hybrid programs to support relational reasoning and direct comparisons of hybrid programs as first-class citizens in the logic. The proof calculus for dRL provides structural methods for proving that hybrid program α refines hybrid program β, written $\alpha \leq \beta$, which simplifies proving by maintaining good proof structure. When a refinement relation has been proved between two hybrid programs, the dRL calculus allows the more restrictive program to automatically inherit all safety properties proved about the program that is more permissive. As a result, dRL enables the use of refinement relations on hybrid programs to reduce a safety argument for α to a safety proof for β. Refinement relations are first-class in the logic dRL, so refinements can be established using reachability properties of hybrid systems.

Differential refinement logic (dRL) combines the ability of *differential dynamic logic* (dL) [19, 21, 22] to reason about the dynamics of hybrid systems with the ability of Kleene algebras for tests (KAT) [13] to relate systems. Unlike dL, dRL provides first-class refinement relations. Unlike KAT, dRL focuses on structural refinement relations as opposed to equivalence relations and dRL works for hybrid systems. And unlike dL as well as KAT, dRL makes it possible to *combine* arguments about the behavior as well as the relation of hybrid systems, which we observe to often be used together for more complex hybrid systems.

Motivating Example. To illustrate dRL's significance, imagine the design of a discrete controller that repeatedly sets a control input u for a continuous system $x' = f(x, u)$ and must remain safely in the region satisfying formula *safe* when started from an initial state satisfying formula *init*. One design paradigm for such controllers is *event-triggered control*, where the controller takes action when cer-

*This material is based upon work supported by National Science Foundation under NSF CAREER Award CNS-1054246. The first author was also supported by a Department of Energy Computational Science Graduate Fellowship, provided under Grant No. DEFG02-97ER25308.

LICS '16, July 05 - 08, 2016, New York, NY, USA

ACM 978-1-4503-4391-6/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2933575.2934555

tain events occur. Formally, this corresponds to adding an evolution domain constraint $E(x)$ to the differential equation $x' = f(x,u)$ that will stop the continuous system and yield control to the discrete system when any of the events have been detected:

$$init \rightarrow [(u \in G(x);\ x' = f(x,u)\ \&\ E(x))^*]\, safe \tag{1}$$

This formula expresses that, if started in *init*, the event-triggered system inside the $[\cdot]$ modality will always satisfy *safe*. This system nondeterministically repeats (operator *) a loop that first sets u arbitrarily according to some given condition $G(x)$ (written $u \in G(x)$) and then (after the ;) follows the differential equation $x' = f(x,u)$. This continuous evolution may evolve for any nondeterministic amount of time within the evolution domain $E(x)$, which stops the continuous system when an event happens. The repetition of the loop then lets the controller respond to such events. Event-triggered control systems such as in (1) are conceptually easy but not implementable faithfully because an implementation would require continuous sensing for these events.

A different design paradigm is *timed-triggered control*, where the controller takes action only every once in a while, after a certain amount of time ε has passed. This corresponds to adding a time-bound $t \leq \varepsilon$ as an evolution domain constraint for a stopwatch t that measures the duration of the current continuous evolution:

$$init \rightarrow [(u := g(x);\ x' = f(x,u)\ \&\ t \leq \varepsilon)^*]\, safe \tag{2}$$

Event-triggered systems such as (1) are significantly easier to verify than time-triggered systems such as (2), which struggle with safely predicting the impact of reaction delays on control decisions. However, only time-triggered systems are implementable.

Without dRL, we would have to choose between an event-triggered paradigm, which is easy to prove, but not a faithful representation of the real system, and the time-triggered paradigm, which is significantly harder to verify. With dRL, we can settle for a proof for the simpler event-triggered system and then subsequently relate it with a refinement proof to the faithful time-triggered system. Essentially, dRL lets us "have our cake and eat it, too."

Now dRL observes that the best way of concluding the time-triggered (2) from a proof of the easier (1) would be to exploit a refinement relation between their systems. An easy way to establish this refinement relation is by separately proving refinement (indicated by $\leq$) of its respective pieces:

$$\begin{array}{c} init \rightarrow [(u \in G(x);\ x' = f(x,u) \& E(x))^*]\, safe \\ \leq\mathrm{I} \qquad\qquad \leq\mathrm{I} \\ init \rightarrow [(u := g(x);\ x' = f(x,u) \& t \leq \varepsilon)^*]\, safe \end{array}$$

This naïve piece-by-piece refinement is great if it works, but, in fact, does not suffice in this case, because *some* knowledge of the invariant properties preserved during the hybrid system run is still needed to establish the refinement. The differential equation of the time-triggered (2) does not refine the event-triggered (1) globally in isolation, but rather only refines it if $g(x)$ has the appropriate behavior of reacting early enough. Hence, dRL provides nested formulas such as $[u := g(x)](\alpha \leq \beta)$, which expresses that α refines β after all runs of $u := g(x)$. dRL provides such behavioral reasoning in support of refinement reasoning. To simplify the verification, the particular discrete controller $u \in G(x)$ in (1) is also more permissive than the easily implementable direct assignment $u := g(x)$ in (2), which showcases another common use of dRL's refinement.

One important design goal for dRL, thus, is that it provides a way of performing refinement reasoning in support of behavioral reasoning: the system in (2) refines the system in (1) such that a proof of behavioral property (1) implies the behavioral property (2). But also that dRL provides a way of performing behavioral reasoning in support of refinement reasoning: the system in (2) refines the system in (1) after taking into account some of the behavioral properties of the respective system runs. dRL, thus, provides a seamless integration of both styles of reasoning by supporting both $[\alpha]\phi$ and $\alpha \leq \beta$ as first-class formulas in the logic that can be nested arbitrarily. Its proof calculus provides ways of using the former shape to justify the latter and vice versa.

2. Syntax

In this section, we introduce *differential refinement logic* (dRL), which adds a refinement relation to differential dynamic logic (dL), a specification and verification language for hybrid systems [21, 22]. Both dL and dRL model cyber-physical systems as *hybrid programs* (HPs). HPs combine differential equations with traditional program constructs and discrete assignments.

Definition 1 (Hybrid program)**.** HPs are defined by the following grammar (where α, β are HPs, x is a variable, θ is a term possibly containing x, and ϕ is a formula of dRL, often first-order):

$$\alpha, \beta\ ::=\ x := \theta \mid x' = \theta\ \&\ \phi \mid ?\phi \mid \alpha \cup \beta \mid \alpha;\beta \mid \alpha^*$$

The effect of *assignment* $x := \theta$ is an instantaneous discrete jump assigning θ to x. The effect of *differential equation* $x' = \theta\ \&\ \phi$ is a continuous evolution where the differential equation $x' = \theta$ holds *and* (written & for clarity) formula ϕ holds throughout the evolution (the state remains in the region described by ϕ).

The effect of *test* $?\phi$ is a *skip* (i.e., no change) if formula ϕ 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* $\alpha;\beta$, HP β starts after α finishes (β never starts if α continues indefinitely). *Nondeterministic repetition* α^* repeats α an arbitrary number of times, including zero times. *Nondeterministic assignment* $x := *$ assigns an arbitrary real number to x and is definable as syntactic sugar for $x' = 1 \cup x' = -1$.

Except for the changes to formulas (addressed in Definition 2), the syntax and semantics of hybrid programs is unchanged from that used by dL [19, 21, 22].

Definition 2 (dRL formula)**.** Formulas in dRL are defined by the following grammar (where ϕ, ψ are dRL formulas, x is a variable, θ_1, θ_2 are terms, and α, β are HPs):

$$\phi, \psi\ ::=\ \theta_1 \leq \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \forall x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \mid \boldsymbol{\beta \leq \alpha}$$

In addition to formulas of first-order real arithmetic, dRL allows formulas of the form $\beta \leq \alpha$ with HPs α and β. Formula $\beta \leq \alpha$ is true in a state ν iff *all* states reachable from that state ν by following the transitions of β could also be reached from state ν by following *some* transitions of α (the transition semantics of HPs are formally defined in Definition 3). Less formally, the behaviors of α from ν subsume those of β, or we say that β *refines* α from the state ν.

Just as in dL, we may write formula $[\alpha]\phi$ with an HP α and a formula ϕ in dRL. Formula $[\alpha]\phi$ is true in a state ν iff formula ϕ is true in all states that are reachable from ν by following the transitions of α. Accordingly, $\langle\alpha\rangle\phi$ is true in a state ν iff formula ϕ is true in some state reachable from ν by α.

The formulas $\beta \leq \alpha$ and $[\alpha]\phi$ make a powerful pair; when both are true, then we know that formula ϕ is true in all states reachable from ν by following the transitions of β, i.e. $[\beta]\phi$. Also since $\beta \leq \alpha$ is not a separate judgment but a formula in dRL, refinement arguments and behavioral arguments can be combined freely. For example, $[\alpha]\phi \wedge (\beta \leq \alpha) \rightarrow [\beta]\phi$ expresses this safety refinement as a dRL formula. And $[\gamma](\beta \leq \alpha)$ expresses that all behavior of β can be mimicked by α from all those states reachable by γ.

3. Semantics

The semantics of HPs are defined as a reachability relation as in dL. A *state* ν is a mapping from the set V of variables to $\mathbb{R}$. The set

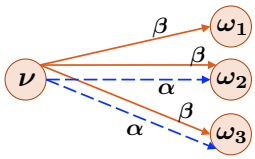

Figure 1: An illustrated example of the transition semantics for HPs α and β where $\nu \models \alpha \leq \beta$. Every state reachable by transitioning on α from ν (in this case, ω_2 and ω_3), is also reachable through β.

of states is denoted $\mathcal{S}$. The value of term θ in state ν is denoted by $[\![\theta]\!]_\nu$.

Definition 3 (Transition semantics of HPs [22])**.** Each HP α is interpreted semantically as a binary reachability relation $\rho(\alpha) \subseteq \mathcal{S} \times \mathcal{S}$ over states, defined inductively and as usual in differential dynamic logic (dL):

- $\rho(x := \theta) = \{(\nu, \omega) \ : \ \omega = \nu$ except that $[\![x]\!]_\omega = [\![\theta]\!]_\nu\}$
- $\rho(?\phi) = \{(\nu, \nu) \ : \ \nu \models \phi\}$
- $\rho(x' = \theta \,\&\, \phi) = \{(\varphi(0), \varphi(r)) \ : \ \varphi(t) \models x' = \theta$ and $\varphi(t) \models \phi$ for all $0 \leq t \leq r$ for a solution $\varphi : [0, r] \to \mathcal{S}$ of any duration $r\}$; i.e., with $\varphi(t)(x') \stackrel{\text{def}}{=} \frac{d\varphi(\zeta)(x)}{d\zeta}(t)$, φ solves the differential equation and satisfies ϕ at all times.
- $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$
- $\rho(\alpha;\beta) = \{(\nu, \omega) : (\nu, \mu) \in \rho(\alpha), (\mu, \omega) \in \rho(\beta)\}$
- $\rho(\alpha^*) = \bigcup_{n \in \mathbb{N}} \rho(\alpha^n)$ with $\alpha^{n+1} \equiv \alpha^n; \alpha$ and $\alpha^0 \equiv ?true$

Definition 4 (dRL semantics)**.** The *satisfaction relation* $\nu \models \phi$ for a dRL formula ϕ in state ν is defined inductively and, aside from the refinement relations, it is defined as in differential dynamic logic. If $\nu \models \phi$, we say that ϕ holds at state ν. A formula ϕ is *valid* iff ϕ holds at all states, i.e. $\nu \models \phi$ for all $\nu \in \mathcal{S}$; we write $\models \phi$ to denote that ϕ is valid. We use ν_x^d to be the state ν with the variable x assigned to real value d.

1. $\nu \models (\theta_1 \leq \theta_2)$ iff $[\![\theta_1]\!]_\nu \leq [\![\theta_2]\!]_\nu$
2. $\nu \models \neg F$ iff $\nu \not\models F$, i.e. if it is not the case that $\nu \models F$
3. $\nu \models F \wedge G$ iff $\nu \models F$ and $\nu \models G$
4. $\nu \models \forall x\, F$ iff $\nu_x^d \models F$ for all $d \in \mathbb{R}$
5. $\nu \models [\alpha]\phi$ iff $\omega \models \phi$ for all ω with $(\nu, \omega) \in \rho(\alpha)$
6. $\nu \models \langle\alpha\rangle\phi$ iff $\omega \models \phi$ for some ω with $(\nu, \omega) \in \rho(\alpha)$
7. $\boldsymbol{\nu \models \alpha \leq \beta}$ iff $\{\omega : (\nu, \omega) \in \rho(\alpha)\} \subseteq \{\omega : (\nu, \omega) \in \rho(\beta)\}$

Differential refinement logic dRL introduces the refinement relation for hybrid programs to the semantics (case 7). The formula $\alpha \leq \beta$ is true in state ν iff the set of all states reachable from ν by following the transitions of HP α is a subset of the states reachable from ν by following the transitions of HP β; see Fig. 1. We also use $\alpha = \beta$ to denote equivalence of hybrid programs α and β, but this is defined syntactically as $\alpha \leq \beta \wedge \beta \leq \alpha$. Thus, $\nu \models \alpha = \beta$ iff α and β have the same (reachability) behavior when starting in state ν.

4. Proof Calculus

A proof calculus associated with a logical language such as dRL is a set of syntactic transformations that are each proved sound. By combining many of these transformations on a complicated formula, we may simplify and break apart the formula until we are left with formulas that are simple enough and can be proved true using quantifier elimination, in which case we have a proof of our original complicated formula. Because this process is entirely syntactic, such a proof can be automatically checked by a computer or even automatically generated by a proof search procedure. In this section we present sequent proof rules for dRL. The semantics of a sequent $\Gamma \vdash \Delta$ is that of the dRL formula $\bigwedge_{\phi \in \Gamma} \phi \to \bigvee_{\psi \in \Delta} \psi$.

$$\frac{\Gamma \vdash [\beta]\phi, \Delta \qquad \Gamma \vdash \alpha \leq \beta, \Delta}{\Gamma \vdash [\alpha]\phi, \Delta}([\leq])$$

$$\frac{\Gamma \vdash \langle\alpha\rangle\phi, \Delta \qquad \Gamma \vdash \alpha \leq \beta, \Delta}{\Gamma \vdash \langle\beta\rangle\phi, \Delta}(\langle\leq\rangle)$$

Figure 2: dRL interaction rules

The defining proof rules of dRL are the interaction rules where modalities and refinements meet; see Fig. 2. The refinement rule for box modalities $[\leq]$ expresses that if formula ϕ holds in every state reachable on β (i.e., $[\beta]\phi$), and α is a refinement of β (i.e., $\alpha \leq \beta$), then ϕ must also hold in every state reachable on α (i.e., $[\alpha]\phi$). Rule $[\leq]$ makes it possible to replace a HP α by β and a refinement proof. Dually, rule $\langle\leq\rangle$ for diamond modalities, which expresses that if α is a refinement of β, and ϕ holds in at least one transition on α, then that same state must also be reachable on β and therefore $\langle\beta\rangle\phi$ must be true, since β contains all behaviors that α can have. Both rules are sound in any sequent context Γ, Δ.

The rules in Fig. 2 use refinements to justify reachability. Conversely, dynamics can play a role in justifying refinements. The rule

$$\frac{\Gamma \vdash \alpha_1 \leq \alpha_2, \Delta \qquad \Gamma \vdash [\alpha_1](\beta_1 \leq \beta_2), \Delta}{\Gamma \vdash (\alpha_1;\beta_1) \leq (\alpha_2;\beta_2), \Delta}(;)$$

proves that the sequential composition $\alpha_1;\beta_1$ refines $\alpha_2;\beta_2$ by showing that α_1 refines α_2 and, after all runs of α_1, that β_1 also refines β_2. It is soundness-critical that the right premise of $(;)$ *is not* $\Gamma \vdash \beta_1 \leq \beta_2, \Delta$, because the assumptions in Γ may no longer hold, since running α_1 may change the state. The flip-side is that rule $(;)$ makes the behavior of α_1 available when showing the refinement $\beta_1 \leq \beta_2$. A global version of rule $(;)$ with the right premise $\vdash \beta_1 \leq \beta_2$ would be sound, but loses all knowledge from the context Γ, Δ, α_1 and is, thus, rarely applicable. Since $\alpha_1 \leq \alpha_2$ by the left premise, rule $[\leq]$ implies that the right premise of rule $(;)$ can soundly be replaced by $\Gamma \vdash [\alpha_2](\beta_1 \leq \beta_2), \Delta$, which gives a weaker derived rule but occasionally makes proving easier. Rule $(;)$ exploits dRL's local semantics of refinement and that dRL allows nested reachability modalities and refinement operators.

The idempotent semiring axioms from KAT [13] as well as unrolling of loops on the left ($unroll_l$) or on the right ($unroll_r$) directly transfer to dRL; see Fig. 3. For hybrid programs, nondeterministic choice $\cup$ is the additive operator, and sequential composition ; is the multiplicative operator. The hybrid program $?\bot$, where $\bot$ is the formula false, is a test that always fails. Its transition semantics $\rho(?\bot)$ is the empty set. So, $?\bot$ is the additive identity (rule $\cup_{id}$) and the multiplicative annihilator (rules $;_{annih-r}$ and $;_{annih-l}$). The hybrid program $?\top$, where $\top$ is the formula true, is a test that always succeeds. It is the multiplicative identity (rules $;_{id-r}$ and $;_{id-l}$), as it does not change the transition semantics of any hybrid program when sequentially composed (i.e. $\rho(?\top)$ is the identity relation).

The first rules in Fig. 4 capture the fact that the refinement relation over hybrid programs is a partial order compatible with $\cup$. The rule $\leq_{trans}$ functions as a cut rule for refinements, while $\leq_{refl}$ is for closing, and $\leq_{antisym}$ relates refinement with its corresponding equivalence relation.[1] Rules $\cup_l$ and $\cup_r$ show refinement separately for choices. The remaining proof rules presented in Fig. 4 take

[1] Equivalence of hybrid programs ($\alpha = \beta$) is syntactic sugar for the conjunction $\alpha \leq \beta \wedge \beta \leq \alpha$, so rule $\leq_{antisym}$ holds by definition.

$$\frac{}{\Gamma \vdash \alpha \cup (\beta \cup \gamma) = (\alpha \cup \beta) \cup \gamma, \Delta}(\cup_{assoc})$$

$$\frac{}{\Gamma \vdash \alpha \cup ?\bot = \alpha, \Delta}(\cup_{id}) \qquad \frac{}{\Gamma \vdash \alpha \cup \beta = \beta \cup \alpha, \Delta}(\cup_{comm})$$

$$\frac{}{\Gamma \vdash \alpha \cup \alpha = \alpha, \Delta}(\cup_{idemp}) \qquad \frac{}{\Gamma \vdash \alpha;(\beta;\gamma) = (\alpha;\beta);\gamma, \Delta}(;_{assoc})$$

$$\frac{}{\Gamma \vdash (?\top;\alpha) = \alpha, \Delta}(;_{id-l}) \qquad \frac{}{\Gamma \vdash (\alpha;?\top) = \alpha, \Delta}(;_{id-r})$$

$$\frac{}{\Gamma \vdash \alpha;(\beta \cup \gamma) = ((\alpha;\beta) \cup (\alpha;\gamma)), \Delta}(dist\text{-}l)$$

$$\frac{}{\Gamma \vdash (\alpha \cup \beta);\gamma = ((\alpha;\gamma) \cup (\beta;\gamma)), \Delta}(dist\text{-}r)$$

$$\frac{}{\Gamma \vdash (?\bot;\alpha) = ?\bot, \Delta}(;_{annih-l}) \qquad \frac{}{\Gamma \vdash (\alpha;?\bot) = ?\bot, \Delta}(;_{annih-r})$$

$$\frac{}{\Gamma \vdash (?\top \cup (\alpha;\alpha^*)) = \alpha^*, \Delta}(unroll_l)$$

$$\frac{}{\Gamma \vdash (?\top \cup (\alpha^*;\alpha)) = \alpha^*, \Delta}(unroll_r)$$

Figure 3: Idempotent semiring axioms from KAT

advantage of structural similarities between hybrid programs. For example, the *unloop* rule allows both hybrid programs α and β to be unrolled simultaneously. It is important that we update the context appropriately by a proper placement of $[\alpha^*]$, as the refinement must hold after any number of loop executions, which we accomplish by requiring that $\alpha \leq \beta$ holds after an arbitrary number of executions of α. Rule *unloop* also makes it possible to transport knowledge of any invariants that hold during α^* to the refinement proof $\alpha \leq \beta$. The $:=*$ proof rule says that assigning x to a specific term θ always refines a program which assigns x nondeterministically to any real value (Example 2 below shows how this rule is applied in the context of a loop, and Example 3 shows extensions to cover guarded nondeterministic assignment). While additional relations exist, we have found the rules listed to be the most crucial structural rules through experience in proving dRL properties.

The dRL rules for loops in Fig. 4 are inspired by KAT but crucially generalized to the presence of sequent contexts. The structural refinement rules $loop_l$ and $loop_r$ exhibit a fundamental asymmetry of refinement with loop initializers ($loop_r$) compared to with loop suffixes ($loop_r$). Unlike in $loop_r$, it is crucial for the soundness of $loop_l$ that we add a $[\alpha^*]$ to its premises.

Fig. 5 illustrates a state-transition diagram for $\alpha^*;\beta$ as well as γ. From the second premise of $loop_r$, β refines γ at the end, for all states reachable through α^* (here represented as ω_3 with γ-successor ω_4), so we know that there is an execution of HP γ such that ω_4 is reachable from ω_3 directly. And from the first premise of $loop_r$, we inductively know that the remaining transitions toward the left of Fig. 6 can also be reached through γ, since $\alpha;\gamma \leq \gamma$ for all states reachable from α^*.

$$\frac{}{\Gamma \vdash \alpha \leq \alpha, \Delta}(\leq_{refl}) \qquad \frac{\Gamma \vdash \alpha \leq \beta, \Delta \quad \Gamma \vdash \beta \leq \gamma, \Delta}{\Gamma \vdash \alpha \leq \gamma, \Delta}(\leq_{trans})$$

$$\frac{\Gamma \vdash \alpha \leq \beta, \Delta \quad \Gamma \vdash \beta \leq \alpha, \Delta}{\Gamma \vdash \alpha = \beta, \Delta}(\leq_{antisym})$$

$$\frac{\Gamma \vdash \alpha \leq \gamma \wedge \beta \leq \gamma, \Delta}{\Gamma \vdash \alpha \cup \beta \leq \gamma, \Delta}(\cup_l) \qquad \frac{\Gamma \vdash \alpha \leq \beta \vee \alpha \leq \gamma, \Delta}{\Gamma \vdash \alpha \leq \beta \cup \gamma, \Delta}(\cup_r)$$

$$\frac{}{\Gamma \vdash (x := \theta) \leq (x := *), \Delta}(:=*) \qquad \frac{\Gamma \vdash \phi \rightarrow \psi, \Delta}{\Gamma \vdash ?\phi \leq ?\psi, \Delta}(?)$$

$$\frac{\Gamma \vdash \alpha_1 \leq \alpha_2, \Delta \qquad \Gamma \vdash [\alpha_1](\beta_1 \leq \beta_2), \Delta}{\Gamma \vdash (\alpha_1;\beta_1) \leq (\alpha_2;\beta_2), \Delta}(;)$$

$$\frac{\Gamma \vdash [\alpha^*](\alpha;\gamma \leq \gamma), \Delta \qquad \Gamma \vdash [\alpha^*](\beta \leq \gamma), \Delta}{\Gamma \vdash \alpha^*;\beta \leq \gamma, \Delta}(loop_l)$$

$$\frac{\Gamma \vdash \beta \leq \gamma, \Delta \qquad \Gamma \vdash (\gamma;\alpha) \leq \gamma, \Delta}{\Gamma \vdash \beta;\alpha^* \leq \gamma, \Delta}(loop_r)$$

$$\frac{\Gamma \vdash [\alpha^*](\alpha \leq \beta), \Delta}{\Gamma \vdash \alpha^* \leq \beta^*, \Delta}(unloop)$$

Figure 4: dRL proof rules

By contrast, rule $loop_l$ does *not* require modalities in the premises, making it very different from its counterpart $loop_l$. Fig. 6 illustrates a state-transition diagram for $\beta;\alpha^*$ as well as γ. By the first premise of $loop_l$, the first β refines γ. And then, by the second premise of $loop_l$, any run of $\gamma;\alpha$ (such as the run from ν via ω_1 to ω_2) can be emulated by just γ since $\gamma;\alpha$ refines γ.

In Fig. 7, we present rules for handling differential equations by refinement. The DC rule says that if a differential equation always evolves within some region H_2 (premise), then this differential equation is equivalent to the same ODE, but with an additional conjunction with formula H_2 as a restriction in the evolution domain (conclusion). This rule is reminiscent of differential cuts [22].

The DR rule says that if two differential equations differ only in their evolution domain, then a refinement relationship is satisfied if the evolution domain of the smaller program is a subset of the evolution domain of the larger program. This rule is reminis-

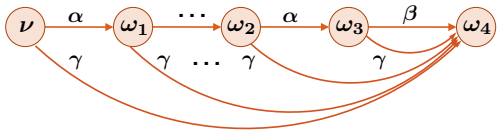

Figure 5: Successive refinement from the right for rule $loop_l$

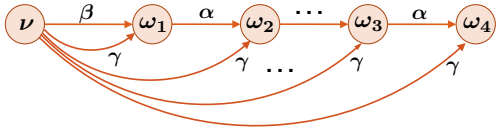

Figure 6: Successive refinement from the left for rule $loop_r$

$$\frac{\Gamma \vdash [x' = \theta \,\&\, H_1]H_2, \Delta}{\Gamma \vdash (x' = \theta \,\&\, H_1) = (x' = \theta \,\&\, H_1 \wedge H_2), \Delta}(DC)$$

$$\frac{\Gamma \vdash \forall x\,(H_1 \rightarrow H_2), \Delta}{\Gamma \vdash (x' = \theta \,\&\, H_1) \leq (x' = \theta \,\&\, H_2), \Delta}(DR)$$

$$\frac{\Gamma \vdash \forall x \Big(\theta\|\eta\| = \eta\|\theta\| \wedge (\|\theta\| = 0 \leftrightarrow \|\eta\| = 0)\Big), \Delta}{\Gamma \vdash (x' = \theta) = (x' = \eta), \Delta}\ (MDF)^2$$

Figure 7: dRL rules for handling differential equations

cent of differential refinements for differential-algebraic dynamic logic [20], where both the evolution domains and the differential-algebraic constraints are compared. In the premise, we quantify universally over the state variables of the hybrid programs $x' = \theta$, so the vector x. Quantifying over x prevents the unsound assumption that the context Γ holds throughout the evolution, when in fact the context is a static property about the initial value of the state x, and is not guaranteed to hold as x evolves.

The *match direction field* (MDF) rule concerns the reachability of two continuous evolutions. Two differential equations are equivalent if they have the same set of reachable states, even if those states are not reached at the same time (unless a clock variable is in the ODE). We quantify over x for similar reasons as in rule DR.

We have proved soundness for MDF for constant differential equations (i.e. in the case where θ does not depend on x) but expect it to generalize without this assumption. The premise implies that the unit direction field[3] of both differential equation systems are identical. In other words, they share all equilibrium points[4], and $\frac{\theta}{\|\theta\|} = \frac{\eta}{\|\eta\|}$ holds everywhere else, where $\|\theta\|$ is the definable Euclidean norm of θ. When the unit direction fields of two differential equations are identical, then their phase portraits[5] are as well. That is, if the two systems start at the same initial value, then the path that their trajectory follows through the state space will be identical. So, while the two systems do not evolve along their trajectories at the same rates, they do have identical sets of reachable states.

Soundness proofs are omitted for length, but are presented in full in the extended version [15, Appendix B].

5. Examples

In the following examples, a $*$ at the top of a proof branch denotes that the branch is closed, either by a proof rule, or by using a decidable algorithm to resolve any remaining $\text{FOL}_{\mathbb{R}}$ properties.

Example 1 ($\alpha \cup \beta = \beta \leftrightarrow \alpha \leq \beta$)**.** In an idempotent semiring, we expect there to be a natural partial order induced by: $\alpha \cup \beta = \beta \leftrightarrow \alpha \leq \beta$, as described in [12]. And, indeed, this rule can be derived for the refinement relation for hybrid programs in dRL as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash \alpha \leq \beta \leftrightarrow \alpha \leq \beta}ax}{\vdash (\alpha \leq \beta \wedge \beta \leq \beta) \leftrightarrow \alpha \leq \beta}\leq_{refl}}{\vdash \alpha \cup \beta \leq \beta \leftrightarrow \alpha \leq \beta}\cup_l}{\vdash (\alpha \cup \beta \leq \beta \wedge \beta \leq \alpha \cup \beta) \leftrightarrow \alpha \leq \beta}\cup_r,\ \leq_{refl}}{\vdash \alpha \cup \beta = \beta \leftrightarrow \alpha \leq \beta}\leq_{antisym}$$

Example 2 (Decomposing a system inside a loop)**.** dRL performs particularly well when determining refinement between HPs which differ only slightly, but within the context of a large and complicated system. In this example, the only difference between the two programs we are comparing is that the program on the left is setting variable x to a specific value θ, and the program on the right allows x to be assigned any value, which is reminiscent of one feature of the example in Section 1 but also of common interest in practice.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash \alpha \leq \alpha} \quad \dfrac{*}{\vdash [\alpha](x := \theta) \leq (x := *)}:=* \quad \dfrac{*}{\vdash [\alpha; x := \theta]\,\beta \leq \beta}}{\vdash (\alpha; x := \theta; \beta) \leq (\alpha; x := *; \beta)};}{\vdash \forall^{(\alpha;x:=\theta;\beta)}(\alpha; x := \theta; \beta) \leq (\alpha; x := *; \beta)}\forall_r}{\vdash [(\alpha; x := \theta; \beta)^*](\alpha; x := \theta; \beta) \leq (\alpha; x := *; \beta)}[]gen}{\vdash (\alpha; x := \theta; \beta)^* \leq (\alpha; x := *; \beta)^*}unloop$$

This is a canonical example of a proof that would be challenging in dL, but is straightforward using refinement. In order to take advantage of the similarities between these two HPs without refinement in dL, we would first have to deal with the outermost operator, here the Kleene star, by finding an appropriate loop invariant, which is necessarily difficult [23]. By contrast, when using dRL, it is possible to be oblivious to the complexities of the two systems where they are the same and focus only on proving refinement in the places where they differ. This is an illustration of *breaking a system into parts*. The quantifiers $\forall^{(\alpha;x:=\theta;\beta)}$ occur after the generalization rule $[]gen$ is applied. This is shorthand notation for universal quantification over all variables bound in the hybrid program $\alpha; x := \theta; \beta$ and over-approximates the reachable states of the hybrid program by allowing the bound variables to take on any value.

Example 3 (Guarded nondeterministic assignment)**.** Example 2 refines a discrete assignment by a nondeterministic assignment. As in the example from Section 1, a more general use case refines a discrete assignment by a guarded nondeterministic assignment $x := *; ?G$ (abbreviated $x \in G$ in Section 1). The nondeterministic assignment $x := *$ assigns an arbitrary real value to x but the subsequent test restricts those values to be the ones satisfying the guard condition G. Using guarded nondeterministic assignment can make a property easier to verify because it has stripped away all the details of the value of x except whatever is necessary for the proof of safety, in this case $G \equiv \phi(x)$.

$$\dfrac{\dfrac{*}{\phi(\theta) \vdash (x := \theta) \leq (x := *)}:=* \quad \dfrac{\dfrac{\dfrac{*}{\phi(\theta) \vdash (\top \rightarrow \phi(\theta))}\rightarrow_r, ax}{\phi(\theta) \vdash (?\top \leq ?\phi(\theta))}?}{\phi(\theta) \vdash [x := \theta](?\top \leq ?\phi(x))}[:=]}{\phi(\theta) \vdash (x := \theta) \leq (x := *; ?\phi(x))};$$

Example 4 (Differential equations)**.** It is often easy to determine whether two atomic hybrid programs are equivalent. For example, $(x := \theta_1) = (x := \theta_2)$ iff $\theta_1 = \theta_2$. However, the same rule does not apply to differential equations. Consider the following formulas:

$$\vdash (x' = 2) = (x' = 9) \tag{3}$$

$$\vdash (x' = 2, t' = 1) \neq (x' = 9, t' = 1) \tag{4}$$

[2] This rule is defined only when no variables of x occur in θ. We use $\|\theta\|$ to indicate the Euclidean norm.

[3] Also known as a slope field, the *direction field* is a graphical representation of a system of differential equations, which plots a vector of the slope of the solution of the differential equation at each point in the state space. A unit direction field is one where each of these vectors has magnitude one.

[4] An *equilibrium point* is a point in the state space where the derivative is zero. More formally, x_0 is an equilibrium point for $\frac{\mathrm{d}x}{\mathrm{d}t} = f(t, x)$ if $f(t, x_0) = 0$ for all t. For linear differential equations, which are polynomial in t, equilibrium points are constant solutions.

[5] A *phase portrait* is the trajectory of a differential equation solution in the state space from a given initial value.

In (3), because the duration of the evolution is nondeterministic, the effect of evolving for an arbitrary duration with a positive derivative is simply that the value of x after evolution is anything greater than or equal to the initial value of x. These two programs differ only in duration, but their reachability relation is the same so long as there is no variable that observes time. However, in (4), time is now being recorded in the variable t. If the two evolutions differ in their duration, the discrepancy is recorded and therefore these programs are not equivalent. The equivalence (3) proves easily:

$$\dfrac{\dfrac{*}{\vdash \forall x.((2 \cdot \|9\| = 9 \cdot \|2\|) \wedge (\|2\| = 0 \leftrightarrow \|9\| = 0))}\,QE}{\vdash (x' = 2) = (x' = 9)}\,MDF$$

Example 5 (Nondeterministic choice and disjunction of tests)**.** It is easy to prove with the dRL calculus that a nondeterministic choice between two tests is equivalent to a single test which joins the two formulas with a disjunction.

$$\Gamma \vdash (?\phi \cup ?\psi) = \ ?(\phi \vee \psi), \Delta$$

We also find a similar dRL proof for an equivalence between sequential composition and conjunction of tests:

$$\Gamma \vdash \ (?\phi; ?\psi) = \ ?(\phi \wedge \psi), \Delta$$

The dRL proofs of these two properties are straightforward, and therefore omitted for space.

Example 6 (Differential cut and refinement)**.** Changing evolution domain constraints of differential equations can have a huge impact on the meaning of the model, because they limit the domain within which the system is allowed to evolve. Yet, under the appropriate initial conditions on x and y, we can prove that the differential equation $x' = y, y' = 1$ will always ensure that $x \geq 0$. As a result, under these initial conditions, adding $x \geq 0$ to the evolution domain does not restrict the set of reachable states, and so the following refinement property is satisfied:

$$x \geq 0 \wedge y \geq 0 \vdash (x' = y, y' = 1) \leq (x' = y, y' = 1 \,\&\, x \geq 0)$$

The dRL proof for this property is in the extended version [15, Appendix A]. The proof shows that $x \geq 0$ is an invariant of the differential equation, which first requires us to show that $y \geq 0$ is also invariant. Once both $x \geq 0$ and $y \geq 0$ have been cut in as invariants, we then use DR to ignore the helper invariant $y \geq 0$ and close the proof.

The DC and DR rules are especially helpful when a differential equation does not have a solution, or when the solution is computationally intractable.

6. Time-Triggered Refines Event-Triggered

Hybrid systems are so called because they have a tight coupling of both discrete components (such as computer controllers) and continuous evolutions. Another source of discrete behavior commonly arises from the measurement of continuous behavior. Continuous values, like position or velocity, are often delivered at discrete time intervals from sensors or via communication. This means that a control decision must be made knowing that updated information may not be available for some time into the future. When the time delay between sensor or communication updates is explicitly modeled, the system is called *time-triggered*. A controller for a time-triggered system would make choices like, "Accelerate at rate a until the next sensor update."

Because time-triggered systems have to make the right choice until the next sensor update, their controllers can be tricky to get right. On top of that, explicitly modeling the sensor delays increases the complexity of the system models. These challenges combine to make time-triggered models tough verification problems. As a result, it is common to make a simplifying assumption that the sensors have continuous access to the values they are measuring. This turns a time-triggered model into an *event-triggered* (also called *event-driven*) model. A controller for an event-triggered system could make choices like, "When the car is 10 feet away from the stop sign, start braking." While this simplification makes modeling and analyzing the behavior of the system easier, continuous sensing is usually not a physical possibility, since it's easy to miss the exact moment when the car is 10 feet away from the stop sign.

This distinction between event-triggered and time-triggered models is an important one [10, 25]. It is a modeling decision that is made early on in the design process and considered a tough one to reverse. However, in this section we compare these models using dRL and, through the lens of refinement, we find that they are formally relatable, and not so fundamentally different after all.

In Section 6.1, we provide a generic model for event-triggered systems. Because their controllers are more directly connected to the physical dynamics through continuous sensing, event-triggered systems are often easier to verify than time-triggered models. From this event-triggered model, a controller for a time-triggered system can be derived which inherits the proof of safety from the event-driven system. This time-triggered model is presented in Section 6.2, and we prove that it refines the event-triggered model using the dRL proof calculus in Section 6.3. We then discuss how to use this refinement relation to simplify the proof of a challenging time-triggered system in Section 6.4.

While we discuss just one case study that builds on the proof of refinement between event-triggered and time-triggered systems, we believe the verification of many time-triggered systems could benefit by also building on this common proof of refinement. Because of the discrete nature of sensors and computers, a vast majority of safety-critical CPS are time-triggered. And many of those operate as a switched system, like the generic model presented in Model 2. In other words, they have a normal operating mode, but after some threshold is reached, they have a discrete switch in behavior (e.g. evading another aircraft or braking to stop in time for a stop light). Many cyber-physical systems will have a collection of such switching conditions, and we leave as an extension of this work proofs for these more complicated systems. Each of these systems can immediately inherit the proof of refinement to show that it refines its event-triggered counter part. We expect that each of these systems would see a similar reduction in the number of required proof steps as the local lane control case study.

6.1 Event-Triggered Model

In this section we introduce Model 1, a generic template for an event-triggered model of a hybrid system. The system may evolve continuously until an event triggers the controller. The controller can then switch to a different mode. For example, if the system is a car and the controlled variable is acceleration, the event could be that the car passes some point on the road at which time the controller immediately switches into braking. The controller can also change the control variable at any time, even before the event trigger, but it has the additional guarantee that it will be able to change the control variable exactly when the event occurs.

In Model 1 we can see that the event-triggered model follows the expected high-level structure: discrete control $ctrl_{Ev}$, followed by the continuous evolution dyn_{Ev}, and then nondeterministically repeat these steps as indicated with * in (5). In (6), we have what is called a guarded nondeterministic assignment to variable u, which first allows u to be set nondeterministically to any value ($u := *$), and then restricts those choices to any that satisfy a formula Safe(x, u). We informally represented this in Section 1 with

Model 1 Event-triggered model

$$\texttt{event}^* \equiv (ctrl_{Ev}; dyn_{Ev})^* \quad (5)$$

$$ctrl_{Ev} \equiv u := c \cup (u := *; ?\text{Safe}(x,u)) \quad (6)$$

$$dyn_{Ev} \equiv t := 0; x_0 := x; \quad (7)$$

$$((x' = f(x,u), t' = 1 \;\&\; E(x) \wedge D(x)) \quad (8)$$

$$\cup (x' = f(x,u), t' = 1 \;\&\; \sim E(x) \wedge D(x))) \quad (9)$$

the notation $u \in \text{Safe}(x,u)$. When $\text{Safe}(x,u)$ is not satisfied, the controller must switch to $u := c$.

We model the continuous dynamics of the system in (8) and (9). The program $x' = f(x,u)$ is a system of differential equations (x may be a vector of several simultaneously evolving variables) that depends on constant control variable u. The evolution domain is $D(x)$, which is some region that the system is not able to evolve beyond and often comes from some physical limit. For example, when we model braking in cars as a deceleration, we have an evolution domain of $v \geq 0$, since braking should not cause the car to start moving backwards. $E(x)$, on the other hand, is the event trigger for the system. When the system reaches the boundary of this domain, the evolution is forced to stop, allowing the controller to execute. After the controller executes, the system must be able to continue evolving, thus the second differential equation in (9). The differential equations and the evolution domain stay the same. However, we use $\sim E(x)$ to represent the topological closure of the complement of $E(x)$. This means that once the event has been passed and the controller executed, the system will still evolve whether or not the controller made a safe choice, making it possible to detect all unsound control choices in the verification step. We require that $E(x)$ be a closed domain so that 1) the evolution can actually reach the event trigger, and 2) the system may transition between (8) and (9) on that boundary.

Variables t and x_0 in (7) are not used anywhere in the program, and are therefore "ghost" variables, since they don't affect the state of the program. However, they do aid in the refinement verification, since they provide an anchor point between Model 1 and Model 2.

6.2 Time-triggered Model

The template for a time-triggered model, presented in Model 2, again loops between a discrete control $ctrl_t$, and continuous evolution dyn_t, as shown in (10). The primary difference between this model and Model 1 is that the time-triggered model can only ensure that the controller will be able to take a control action at least every ε seconds. This is expressed in line (13) by first setting clock t to zero, and then evolving continuously along the differential equations with $t' = 1$, but only within the evolution domain of $t \leq \varepsilon$. Notice that the event trigger $E(x)$ is *not* in the evolution domain for the differential equation in the time-triggered model, so the controller's conditions are only checked sporadically.

Model 2 Time-triggered model

$$\texttt{time}^* \equiv (ctrl_t; dyn_t)^* \quad (10)$$

$$ctrl_t \equiv u := c \cup (u := *; ?\text{Safe}_\varepsilon(x,u)) \quad (11)$$

$$dyn_t \equiv t := 0; x_0 := x; \quad (12)$$

$$(x' = f(x,u), t' = 1 \;\&\; t \leq \varepsilon \wedge D(x)) \quad (13)$$

Another major difference between the time-triggered model and the event-triggered model is in the discrete controller. Because the discrete controller in the time-triggered model must make a choice that will be safe for up to ε time, we have to change the guard on the nondeterministic assignment of control variable u in (11). The guard $\text{Safe}_\varepsilon(x,u)$ depends both on the current choice of u and the time duration ε, in addition to the current state x.

6.3 Proof of Refinement

We expect any safe controller of the time-triggered model to be more conservative than even the most admissible (but still safe) controller for the event-triggered model. In other words, we expect the time-triggered model to not have as wide a range of control choices as the event-triggered model. This is because the event-triggered model has continuous access to sensor data, while the time-triggered model only samples it discretely. As a result, the set of possible behaviors of the more conservative time-triggered controller are expected to be a subset of the possible behaviors of the event-triggered model. More formally, we expect the time-triggered model to refine the event-triggered model. This is great news for us, since generally speaking, event-triggered models are far easier to verify, but time-triggered models are more reasonable to implement. Now all we have to do is take advantage of this refinement relation in the proof structure using dR$\mathcal{L}$ refinement proof rules.

For example, suppose that we want to implement a time-triggered system that always satisfies some safety condition, ϕ. We write the condition that our time-triggered model, $\texttt{time}^*$, always satisfies ϕ using the box modality: $[\texttt{time}^*]\phi$. We would first apply the $[\leq]$ rule, as below, to split the property into two sub-goals. First, that an event-triggered model satisfies the safety condition, $[\texttt{event}^*]\phi$. And second, that the original time-triggered model refines the event-triggered model, $\texttt{time}^* \leq \texttt{event}^*$.

$$\frac{\Gamma \vdash [\texttt{event}^*]\phi, \Delta \qquad \Gamma \vdash (\texttt{time}^* \leq \texttt{event}^*), \Delta}{\Gamma \vdash [\texttt{time}^*]\phi, \Delta}\ [\leq]$$

Recall that $\texttt{event}^*$ and $\texttt{time}^*$ are generic templates for event- and time-triggered systems. Without concrete choices for example for the controllers $ctrl_{Ev}$ and $ctrl_t$, it will not be possible to close this proof. However, we can use dR$\mathcal{L}$ proof rules independently from the concrete controllers to significantly simplify the remaining open goals. This proof is presented in full in [14, Chapter 6.3], but we give a proof sketch in this section and outline the open goals that will remain to be proved when concrete models are plugged in.

In this proof, we assume that the event trigger is also an invariant for $\texttt{event}^*$. This is a reasonable assumption to make, since the event trigger can be thought of as the last possible moment when switching to the control choice will still guarantee safety for the system. Consider the simple example of an event-triggered controller for a car, where the car applies the brakes 10 feet before a stop sign. This means that we define $E(x) \equiv d \geq 10$, where d is the distance between the car and the stop sign. This event trigger will not be an invariant for the system, since the car will pass the 10 feet away mark after it starts braking. But also notice that this event trigger doesn't take into account the velocity of the car. If the car is traveling fast enough, braking 10 feet away from the stop sign may not be enough distance for the car to stop in time.

Instead, a better (and provably safe) event trigger will be symbolically defined. By defining the event trigger to be the last possible moment when the car can brake and not run the stop sign, we get $E(x) \equiv d \geq \frac{v^2}{2B}$, where d again is the distance between the car and the stop sign, v is the car's velocity, and B is the braking force that the car engages when the event trigger occurs. In this case, when the car hits the boundary of $E(x)$ and starts to brake, it then evolves along the boundary of $E(x)$, since while the distance to the stop sign decreases, so does the velocity of the car. This makes $E(x)$ an invariant of the system in this particular case, but also demonstrates why this assumption is often satisfied, as event triggers are usually invariants of the system. This same system invariant should

actually be proved invariant within the proof of safety for the event-triggered model (the canonical proof style for this property), so that it may be reused in the proof of ($\texttt{time}^* \leq \texttt{event}^*$).

We require for this proof that the solution to the differential equation exists and is expressible as a term in dRL. We define $S_{x_0,u}(t)$ to be the solution to $x' = f(x,u)$ at time t, with x_0 as the initial value of state x.

In the proof that $\texttt{time}^*$ refines $\texttt{event}^*$, three open goals remain to be proved based on concrete implementation specifics. Note that none of these goals contains a differential equation or any hybrid programs! These three open goals are all expressed in the decidable first-order logic over the reals ($\text{FOL}_{\mathbb{R}}$).

Open Goal 1 - Discrete Controllers Satisfy Refinement:

$(\Gamma \wedge E(x) \wedge \text{Safe}_\varepsilon(x,u)) \vdash \text{Safe}(x,u)$

This open goal appears in the proof branch where we show that the discrete controllers satisfy the refinement relationship, which depends on the specific choices of $\text{Safe}(x,u)$ and $\text{Safe}_\varepsilon(x,u)$. This open goal requires that until reaching the event trigger, being safe for ε time ($\text{Safe}_\varepsilon(x,u)$) implies $\text{Safe}(x,u)$.

Open Goal 2 - Evade Mode:

$(\Gamma \wedge E(x_0) \wedge 0 \leq t \leq \varepsilon \wedge \overline{x} = S_{x_0,c}(t) \wedge D(\overline{x})) \vdash E(\overline{x})$

This open goal appears in the proof branch where we show that the continuous dynamics satisfy refinement. It requires that the control choice $u := c$ is enough to ensure that the system will not cross the event-trigger boundary within time ε. In other words, if the invariant/event trigger is initially satisfied (i.e. $E(x_0)$ is satisfied), then for all time t between 0 and ε, the solution at time t (named $\overline{x}$) must satisfy the invariant (i.e. $E(\overline{x})$).

Open Goal 3 - Normal Mode:

$(\Gamma \wedge E(x_0) \wedge \text{Safe}_\varepsilon(x_0,\overline{u}) \wedge 0 \leq t \leq \varepsilon \wedge \overline{x} = S_{x_0,\overline{u}}(t) \wedge D(\overline{x})) \vdash E(\overline{x})$

This open goal is similar to Open Goal 2, except that the control choice is nondeterministic ($u := *$), since we are in normal mode rather than evade. We represent this control value with the variable $\overline{u}$. Of course, we may only choose $\overline{u}$ if it satisfies $\text{Safe}_\varepsilon(x_0,\overline{u})$. This goal requires that if the invariant/event trigger is initially satisfied (i.e. $E(x_0)$), and the guard was satisfied initially for any choice of acceleration $\overline{u}$ (i.e. $\text{Safe}_\varepsilon(x_0,\overline{u})$ is satisfied), then for all time t between 0 and ε, the solution at time t (named $\overline{x}$) must satisfy the invariant (i.e. $E(\overline{x})$).

Because these three open goals are expressions in first-order logic over the reals, a decidable fragment of dRL, these open goals are usually much easier to verify. The full proof of refinement, and a detailed discussion of proving techniques can be found in [14, Chapter 6.3]. The proof requires 50 steps and crucially leverages the localness of refinement (i.e. properties with mixed box and refinement formulas) to close several branches that depend on partial knowledge of the program context in which their refinements occur.

When using theorem proving to verify hybrid systems, it is not uncommon to first prove safety for an event-triggered model of the system, and then add in modeling of a time delay and reprove safety for the time-triggered model [11]. Event-triggered models are easier to prove because they avoid the complications introduced by delays and reaction times that are modeled in time-triggered architectures. Now, instead of reproving from nothing, the proof of safety for the time-triggered system can be built on top of the proof of safety for the event-triggered system once and for all.

While this refinement proof is somewhat involved, the same proof can be immediately reused for any time-triggered or event-triggered systems that fit the generic templates.

6.4 Case Study: Local Lane Control

In [16], a time-triggered model for local lane control (llc) is defined and verified using the dL proof calculus and associated theorem prover KeYmaera. The proof verifies collision freedom for an adaptive cruise control system for two cars driving on a straight lane. However, proving this safety property using the dL proof calculus in KeYmaera required enormous effort: 656 interactive proving steps. The proof of safety for the event-triggered model for the same system required just 4 interactive steps [14, Chapter 6.4].

Now that we have a template for a time-triggered system that provably refines an event-triggered system, we can revisit the proof of safety for llc to see if it could have been completed with less effort. In order to directly use the template in Model 2 for the local lane control system, we make a few modifications from the original model presented in [16]. First, we add the ghost variable x_0 to keep track of the initial value of x before each continuous evolution. Next, we assume that when a car applies the brakes, it does so with exact braking power $-B$, rather than nondeterministically within some range. This is because our proof of refinement in Section 6.3 uses a deterministic control choice for the evasive maneuver branch. We leave it as future work to extend the refinement proof to allow nondeterministic controllers in both the normal and evasive modes. And finally, the controller in [16] has an additional control branch which allows the car to remain stopped once it brakes to a stop. We remove this branch because the proof of refinement in Section 6.3 only allows two control modes: normal and evade. We can argue informally that a stopped car does not pose a risk of colliding with a car in front of it. Each of these changes still constitute reasonable representations of the underlying system, however they lessen the impact of direct comparisons of proof statistics between the two models.

To verify the adapted time-triggered llc model, all that remains is to close the three open goals. This is done easily in KeYmaera with Goals 1 and 2 closing automatically, and Goal 3 requiring 79 interactive steps. Including the proof of the event-/time-triggered refinement property and the event-triggered proof, the total number of interactive steps required to prove safety for this adapted time-triggered model is 133, an 80% reduction. Computation time also decreased by 74%. The full models, as well as links to all electronic proofs can be accessed in [14, Chapter 6.4].

While there are some differences in the structure of the compared models (stated above) that complicate direct comparisons, these are significant improvements over the original proof in dL, and should be considered strong evidence that dRL can reduce the number of user interactions and computation required for proving.

7. Relating Differential Refinement Logic

Every instance of the refinement relation that dRL adds to dL can be defined equivalently in dL, so we could easily lift the axioms from dL and add the equivalence transformation to accomplish completeness for dRL based on the relative completeness of dL [22]:

$$\models_{\text{dRL}} \alpha \leq \beta \iff \models_{\text{dL}} \forall \bar{x}\, (\langle\alpha\rangle(x = \bar{x}) \rightarrow \langle\beta\rangle(x = \bar{x})) \qquad (14)$$

where x is a vector of all bound variables in either α or β, and $\bar{x}$ is a vector of fresh variables of equal length to x. While this observation gives us an easy out for completeness, converting refinement into a dL property in this way would completely undermine dRL's goal – taking advantage of the structure of hybrid programs to prove refinement relations. In practice, the formula on the right-hand side of (14) will be significantly more complicated to verify (due to the added diamond modalities and quantifying over variables) than verifying properties about α and β directly, making this a

terrible idea in practice. Additionally, (14) can only be expressed in dL separately for each concrete pair of α, β, but dRL provides refinement as one operator for all α, β.

The canonical goal of dRL is to be able to statically verify safety for a hybrid program α by showing that it refines a verified system β. What we often gain with dRL is a verified system α that more accurately models the real-world system than β. Yet where the abstractions and generalizations of β make β easier to verify.

Just as it is possible to rewrite refinement statements using dL, it is also possible to, vice versa, encode the box modality of dL purely as a refinement relation:

$$\models_{\mathsf{dL}} [\alpha]\phi \iff \models_{\mathsf{dRL}} \alpha \leq (x := *; ?\phi) \tag{15}$$

where x again is a vector of all bound variables in α, and $x := *$ is a nondeterministic assignment. The HP $x := *; ?\phi$ transitions from some starting state ν to any state that may differ from ν only on the bound variables of α, here x, and in which ϕ is satisfied.

Lemma 1 (Expressiveness)**.** *dRL and dL are equally expressive: every formula in one logic has a formula in the other logic that is equivalent. This remains true when dropping modalities from* dRL*.*

Proof. dL is a fragment of dRL. dRL provides the extension of adding $\alpha \leq \beta$ as a logical formula, which is definable by an equivalent dL formula according to (14). To show that dRL without modalities can express all dL formula, we show by induction that all modal formulas $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ of dL can be expressed in dRL without modalities. By induction hypothesis, ϕ can be assumed to have been replaced by an equivalent without modalities already. For the formula $[\alpha]\phi$, (15) gives an equivalent encoding in dRL. For the formula $\langle\alpha\rangle\phi$, which is equivalent to $\neg[\alpha]\neg\phi$, the dRL formula $\neg(\alpha \leq x := *; ?\neg\phi)$ is an equivalent encoding. □

These encodings illustrate that the value of dRL is not in an increased expressiveness or in completeness considerations, but rather in the practical value that its additional proof structure of hybrid system relations enables. dRL makes proofs with mixed behavioral and refinement arguments possible and natural that would otherwise be impossible or only emulated with encodings that make the proof worse or with significant effort in proof planning.

8. Related Work

Refinement and discrete programs. Kleene algebra with tests make it possible to manipulate programs that are equivalent [13]. Hybrid programs in dL form an idempotent semiring when you take sequential composition as the multiplicative operator, and nondeterministic choice as an additive operator. Adding in the Kleene star and the test gives us a Kleene algebra with tests (KAT). As a result, we draw heavily on research done on these algebras when designing the corresponding proof calculus for dRL. The proof rules presented in Fig. 3 are derived directly from KAT axioms. But this is not the end of the story for dRL, as we still have to handle the complexities of assignments and differential equations. Even simple hybrid programs that would seem trivially unrelated when examined through the lens of KAT, may in fact satisfy the refinement relation. For example, consider the hybrid programs $x := 10$ and $x := 1; x' = x$. The program $x := 10$ assigns the value 10 to variable x. The program $x := 1; x' = x$ first sets x to 1 and then follows the solution to the differential equation $x' = x$ for a nondeterministic amount of time. While these hybrid programs appear unrelated, the first hybrid program is actually a refinement of the second. The continuous evolution that follows $x' = x$ evolves for a nondeterministic period of time, thus allowing x to take any value greater than 1, which means it includes a transition where x takes value 10.

Moreover in formal methods for cyber-physical systems, it is critical to express a refinement relationship between two programs, since we are always trying to refine the programs we verify into programs that more closely represent the real conditions in which they operate. While KAT has rules for handling refinement behaviors, its focus is on manipulating equivalent programs. Finally, dRL leverages the interplay of dL's modalities for proving properties of system dynamics with refinement relations on programs.

We present in dRL a local refinement relation, which can take advantage of the context and the surrounding hybrid program to prove refinement; several related research areas explore the notion of global refinement [2, 6, 7, 13, 26]. This difference is most striking for sequential composition. Under a global definition of refinement, if $\alpha_1 \leq \alpha_2$ and $\beta_1 \leq \beta_2$, then $\alpha_1;\beta_1 \leq \alpha_2;\beta_2$. However, we often want to prove refinement of subsystems within larger programs. These subsystems usually do not satisfy global refinement, but do satisfy local refinement only within the particular context of the larger surrounding system. Some notions of local refinement may be recovered by augmenting with guards and asserts [2, 6, 7]; however, this requires defining a formula to represent the exact set of states reachable through α_1. Coming up with such a formula is often very difficult, particularly in hybrid systems where values evolve continuously over time, and may require over-approximation.

Refinement and dL. One major challenge for formal verification of cyber-physical systems is that there will always be a gap between the behavior of a statically verified model and the behavior of the physical implementation of the system "in the wild." This is because the many continuous environment and state variables can never be fully captured and precisely represented. While dRL reduces this gap by making more challenging models verifiable, it will not be able to fully verify a system against circumstances that are not explicitly represented in the model. However, ModelPlex[17] implements a run-time analysis that samples the observed state of a real system via sensors and checks in real time that this state refines the reachable states of the statically verified model. In fact, ModelPlex itself is predicated on a study of a refinement relation, which dRL lifts to the level of a logic.

Semantic versions of refinement relationships are also at the heart of an approach for a library of proof-aware refactoring operations [18] for hybrid programs in dL, which make it possible to change a model with maximal reuse of its safety and/or liveness proof. While these refactoring transformations have been shown to be effective, each refactoring pattern has to be justified by a semantical argument from scratch. dRL takes the more fundamental approach of including refinement as a first-class citizen into the logic and developing a set of proof rules once and for all that can be used to formally prove any such refinement or pattern formally. Additionally, we examine refinement within a given logical context, so dRL supports refinements $\alpha \leq \beta$ that only hold under the current context while not holding generally.

Refinement and hybrid systems While the refinement relation as a first-class member of dRL is new, the concept of refinement has been in use for quite some time. Discrete model checking, for example, has seen tremendous success in using abstraction to keep the statespace small, then iteratively refining the model to exclude spurious counter examples (CEGAR [5]).

Refinements are the primary development step in Event-B. Recent work by Banach et al. introduces Hybrid Event-B [3], which adds continuous variables with continuous evolution of those variables over time intervals to the Event-B framework. Butler et al. define a restricted notion of refinement for Hybrid Event-B [4] which has a two-pronged approach: reducing nondeterminism in an abstract continuous evolution of the system (ignoring its differential equation) and adding additional discrete actions to a model. dRL handles both kinds of refinements as special cases. While abstractions of differential equations are supported in dRL, its refinement

properties are not just assumed but actually proved in dRℒ via differential invariants, differential cuts, and differential refinements.

Approximate bisimulations are another approach for relating two continuous systems [8, 9]. Unlike exact bisimulation, which requires two systems to be identical under a mapping between them, approximate bisimulation only requires that the systems be close. If a bound can be calculated for the maximal error for the approximate system, then the original system can be proved safe if the approximate system is safe with the maximal error as extra safety margin. In contrast, dRℒ's working principle is compositional by allowing several local reasoning steps about parts of a system to support a refinement argument. It also gives a general way of combining behavioral and refinement arguments. Applying some of the underlying concepts of approximate bisimulation to dRℒ could result in an interesting extension where state variables are allowed to be fuzzed by some bounded margin of error.

9. Conclusion and Future Work

This paper presents *differential refinement logic* (dRℒ), a specification and verification logic that allows the proof of properties of hybrid programs as well as the direct comparison of hybrid programs. We present a proof calculus for dRℒ and prove it sound. The rules in the proof calculus can be partitioned into three types: 1) structural proof rules, which leverage structural similarities between hybrid programs, 2) proof rules for differential equation refinements, and 3) rules based on the axioms of Kleene algebra with tests.

As an application of dRℒ, we also present the first formal proof that a generic time-triggered system refines its event-triggered counterpart. This refinement relation is an important one, as it ties together two architecture types that are classically considered fundamentally different and incompatible modeling choices. Event-triggered systems are generally considered easier to verify, while time-triggered systems give a more faithful representation of real-world systems with discrete sensors. By establishing this relationship between the two, we can get the best of both worlds. We first prove properties about event-triggered systems which are significantly easier to verify. By proving a few simple side conditions, the resulting proofs extend easily in dRℒ to verify the time-triggered systems, which give a more realistic model of discrete sensors.

While this paper introduces the foundations of the dRℒ logic and provides solid evidence that dRℒ can simplify many challenges of verification for hybrid systems, there are still many questions to be explored, and in particular with a view to automation: How can an automated or semi-automated proof search take advantage of the added proof structure that dRℒ provides? Can refinement aid automatic synthesis for verified implementation of hybrid programs?

In conclusion, dRℒ improves the feasibility of theorem proving for hybrid systems by making it easier to break systems into smaller subsystems, abstract implementation-specific design details, leverage an iterative approach to system design, and maintain a modular proof structure, each by making refinements explicit in the proof.

References

[1] R. Alur. Formal verification of hybrid systems. In *2011 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 273–278, 2011.

[2] R.-J. J. Back, A. Akademi, and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1st edition, 1998.

[3] R. Banach, H. Zhu, W. Su, and R. Huang. Continuous KAOS, ASM, and formal control system design across the continuous/discrete modeling interface: a simple train stopping application. *Formal Aspects of Computing*, 26(2):319–366, 2014.

[4] M. Butler, J.-R. Abrial, and R. Banach. Modelling and refining hybrid systems in Event-B and Rodin. In L. Petre and E. Sekerinski, editors, *From Action System to Distributed Systems: The Refinement Approach*. Taylor & Francis, 2015.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[6] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic (TOCL)*, 7(4):798–833, 2006.

[7] T. Ehm, B. Möller, and G. Struth. *Kleene modules*. Springer, 2003.

[8] A. Girard and G. J. Pappas. Approximate bisimulation: A bridge between computer science and control theory. *European Journal of Control*, 17(5):568–578, 2011.

[9] A. Girard, A. A. Julius, and G. J. Pappas. Approximate simulation relations for hybrid systems. *Discrete Event Dynamic Systems*, 18(2): 163–179, 2008.

[10] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, pages 86–101. Springer, 1991.

[11] Y. Kouskoulas, D. W. Renshaw, A. Platzer, and P. Kazanzides. Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In C. Belta and F. Ivancic, editors, *HSCC*, pages 263–272. ACM, 2013.

[12] D. Kozen. *The design and analysis of algorithms*. Springer, 1992.

[13] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.

[14] S. M. Loos. Differential refinement logic. Technical Report CMU-CS-15-144, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Feb 2016.

[15] S. M. Loos and A. Platzer. Differential refinement logic. Technical Report CMU-CS-16-111, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2016.

[16] S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 42–56. Springer, 2011.

[17] S. Mitsch and A. Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 2016.

[18] S. Mitsch, J.-D. Quesel, and A. Platzer. Refactoring, refinement, and reasoning: A logical characterization for hybrid systems. In *FM*, volume 8442 of *LNCS*, pages 481–496. Springer, 2014.

[19] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. ISSN 0168-7433.

[20] A. Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.*, 20(1):309–352, 2010. ISSN 0955-792X.

[21] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. ISBN 978-3-642-14508-7.

[22] A. Platzer. Logics of dynamical systems. In *LICS*, pages 13–24, 2012.

[23] A. Platzer. Differential game logic. *ACM Trans. Comput. Log.*, 17(1): 1:1–1:51, 2015. ISSN 1529-3785.

[24] A. Platzer. A uniform substitution calculus for differential dynamic logic. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 467–481. Springer, 2015.

[25] F. Scheler and W. Schröder-Preikschat. Time-triggered vs. event-triggered: A matter of configuration? In *MMB Workshop Proceedings GI/ITG Workshop on Non-Functional Properties of Embedded Systems*, pages 1–6. VDE, 2006.

[26] J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51(1):23–45, 2004.

[27] W. Walter. In *Ordinary Differential Equations*, pages 105–157. Springer, 1998.