Refactoring, Refinement, and Reasoning
A Logical Characterization for Hybrid Systems

Stefan Mitsch, Jan-David Quesel, and André Platzer

Computer Science Department
Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. Refactoring of code is a common device in software engineering. As cyber-physical systems (CPS) become ever more complex, similar engineering practices become more common in CPS development. Proper safe developments of CPS designs are accompanied by a proof of correctness. Since the inherent complexities of CPS practically mandate iterative development, frequent changes of models are standard practice, but require reverification of the resulting models after every change.
To overcome this issue, we develop *proof-aware refactorings for CPS*. That is, we study model transformations on CPS and show how they correspond to relations on correctness proofs. As the main technical device, we show how the impact of model transformations on correctness can be characterized by different notions of refinement in differential dynamic logic. Furthermore, we demonstrate the application of refinements on a series of safety-preserving and liveness-preserving refactorings. For some of these we can give strong results by proving on a meta-level that they are correct. Where this is impossible, we construct proof obligations for showing that the refactoring respects the refinement relation.

1 Introduction

Cyber-physical systems combine discrete computational processes with continuous physical processes (e. g., an adaptive cruise control system controlling the velocity of a car). They become increasingly embedded into our everyday lives while at the same time they become ever more complex. Since many CPS operate in safety-critical environments and their malfunctioning could entail severe consequences, proper designs are accompanied by a proof of correctness [2]. The inherent complexity of CPS practically mandate iterative development with frequent changes of CPS models. With current formal verification methods, however, these practices require reverification of the resulting models after every change.

To overcome this issue, we develop proof-aware refactorings for CPS. Refactorings are systematic changes applied to a program or model, and a common method in classical software engineering. In the classical sense [17], refactorings transform the structure of a program or model without changing its observable behavior. Regression testing is a common mechanism used to establish some confidence in the correctness of a classical refactoring [8]. In the presence of

correctness proofs, however, we can analyze refactoring operations w.r.t. refinement of models and their effect on the proven correctness properties. This gives unquestionable, and thus significantly stronger, evidence than regression testing and allows us, moreover, to actually change the observable behavior (e. g., improve energy-efficiency of a controller) while preserving correctness properties.

As the main technical device, we show how the impact of model transformations on correctness can be characterized in differential dynamic logic (dℒ) [18,20]. We present different notions of refinement and prove that they can be logically characterized in dℒ. There are many different ways to refine models (e. g., trace refinement [9], abstraction refinement [6]); we focus on refinement w.r.t. the reachable states, which allows us to transfer correctness properties but still leaves sufficient flexibility to modify the behavior of a CPS. Furthermore, we demonstrate the application of refinements to define a series of safety-preserving and liveness-preserving refactorings. For some refactorings we give strong results by proving on a meta-level that they are correct unconditionally. For those refactorings where correctness cannot be shown on a meta-level, we construct proof obligations based on the logical characterization of our refinement notion in dℒ, which imply that the refactoring respects the refinement relation. Hence these can be conveniently discharged using our existing theorem prover KeYmaera [22].

2 Related Work

Counterexample guided abstraction refinement (CEGAR [6]) uses abstraction to keep the state space in model checking small, and refines the state space abstraction when spurious counterexamples are found. Similar approaches have been suggested for reachability analysis of hybrid systems (e. g., [5,7]).

The Rodin tool [1] enables users to perform refactorings on Event-B [1] models and generates proof obligations where necessary in order to establish refinement between the models. Although originally defined without a concrete semantics in mind these proof obligations have recently been given a solid semantics [25] in terms of CSP [9]. The formal refinement notion used in their approach is based on trace inclusion [9]. The notion of trace refinement is also used to enable compositional modeling of hierarchical hybrid systems in CHARON [3]. Similarly, Tabuada [26] studies refinements based on behaviors of systems that are characterized by their traces. In [4] refinements in the setting of abstract state machines are considered. However, their definition of refinement is a form of simulation relation, which is even stronger than trace inclusion. In contrast, we study notions of refinement based on reachable states. Thus, we are more flexible in terms of which systems we consider to be refinements of each other.

In software development, extensive catalogs of refactoring operations were proposed (e. g., [8,17]). These refactorings, however, target solely the structure of a program to make it easier to understand and maintain, and do not use formal verification to show the correctness of refactoring.

In formal verification of robotic systems [11,12,15] refactoring is used intuitionally to introduce more realistic assumptions into a model that was ini-

tially simplified for correctness verification. For example, in [12] an initial event-triggered model is transformed into a time-triggered model; in [11] duplicated program fragments are removed; in [13] sensor uncertainty is added. After the refactoring, all correctness properties were reverified on the refactored model.

3 Refinement, Refactoring and Proof Obligations

3.1 Preliminaries: Differential Dynamic Logic

Syntax and informal semantics. For specifying and verifying correctness statements about hybrid systems, we use *differential dynamic logic* ($\mathsf{d}\mathcal{L}$) [18,20], which supports *hybrid programs* as a program notation for hybrid systems. The syntax of hybrid programs is generated by the following EBNF grammar:

$$\alpha ::= \alpha;\beta \mid \alpha \cup \beta \mid \alpha^* \mid x := \theta \mid x := * \mid x_1' = \theta_1, \ldots, x_n' = \theta_n \;\&\; H \mid ?\phi \ .$$

The sequential composition $\alpha;\beta$ expresses that β starts after α finishes. The non-deterministic choice $\alpha \cup \beta$ follows either α or β. The non-deterministic repetition operator α^* repeats α zero or more times. Discrete assignment $x := \theta$ instantaneously assigns the value of the term θ to the variable x, while $x := *$ assigns an arbitrary value to x. $x' = \theta \;\&\; H$ describes a continuous evolution of x within the evolution domain H. The test $?\phi$ checks that a particular condition expressed by ϕ holds, and aborts if it does not. A typical pattern $x := *; ?a \leq x \leq b$, which involves assignment and tests, is to limit the assignment of arbitrary values to known bounds. Note that control flow statements like `if` and `while` can be expressed with these primitives [18].

To specify correctness properties about hybrid programs, $\mathsf{d}\mathcal{L}$ provides modal operators $[\alpha]$ and $\langle\alpha\rangle$. When ϕ is a $\mathsf{d}\mathcal{L}$ formula describing a state and α is a hybrid program, then the $\mathsf{d}\mathcal{L}$ formula $[\alpha]\phi$ expresses that all states reachable by α satisfy ϕ. Dually, $\mathsf{d}\mathcal{L}$ formula $\langle\alpha\rangle\phi$ expresses that there is a state reachable by the hybrid program α that satisfies ϕ. The set of $\mathsf{d}\mathcal{L}$ formulas is generated by the following EBNF grammar (where $\sim \in \{<, \leq, =, \geq, >\}$ and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot, /$ over the reals):

$$\phi ::= \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi \mid \forall x \phi \mid \exists x \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \ .$$

Formal semantics. The semantics of $\mathsf{d}\mathcal{L}$ is a Kripke semantics in which states of the Kripke model are states of the hybrid system. Let $\mathbb{R}$ denote the set of real numbers. A state is a map $\nu : \Sigma \rightarrow \mathbb{R}$; the set of all states is denoted by $\mathrm{Sta}(\Sigma)$. We write $\nu \models \phi$ if formula ϕ is true at state ν (Def. 2). Likewise, $[\![\theta]\!]_\nu$ denotes the real value of term θ at state ν. The semantics of HP α is captured by the state transitions that are possible by running α. For continuous evolutions, the transition relation holds for pairs of states that can be interconnected by a continuous flow respecting the differential equation and invariant region. That is, there is a continuous transition along $x' = \theta \,\&\, H$ from state ν to state ω, if there is a solution of the differential equation $x' = \theta$ that starts in state ν and ends in ω and that always remains within the region H during its evolution.

Definition 1 (Transition semantics of hybrid programs). *The* transition relation ρ *specifies which state* ω *is reachable from a state* ν *by operations of* α*. It is defined as follows.*

1. $(\nu,\omega) \in \rho(x := \theta)$ *iff* $[\![z]\!]_\nu = [\![z]\!]_\omega$ *f.a.* $z \neq x$ *and* $[\![x]\!]_\omega = [\![\theta]\!]_\nu$.
2. $(\nu,\omega) \in \rho(x := *)$ *iff* $[\![z]\!]_\nu = [\![z]\!]_\omega$ *f.a.* $z \neq x$.
3. $(\nu,\omega) \in \rho(?\phi)$ *iff* $\nu = \omega$ *and* $\nu \models \phi$.
4. $(\nu,\omega) \in \rho(x_1' = \theta_1, \ldots, x_n' = \theta_n \,\&\, H)$ *iff for some* $r \geq 0$*, there is a (*flow*) function* $\varphi{:}[0,r] \to \mathrm{Sta}(V)$ *with* $\varphi(0) = \nu, \varphi(r) = \omega$*, such that for each time* $\zeta \in [0,r]$*: (i) The differential equation holds, i.e.,* $\frac{\mathsf{d}[\![x_i]\!]_{\varphi(t)}}{\mathsf{d}t}(\zeta) = [\![\theta_i]\!]_{\varphi(\zeta)}$ *for each* x_i*. (ii) For other variables* $y \notin \{x_1, \ldots, x_n\}$ *the value remains constant, i.e.,* $[\![y]\!]_{\varphi(\zeta)} = [\![y]\!]_{\varphi(0)}$*. (iii) The invariant is always respected, i.e.,* $\varphi(\zeta) \models H$.
5. $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$
6. $\rho(\alpha;\beta) = \{(\nu,\omega) : (\nu,z) \in \rho(\alpha), (z,\omega) \in \rho(\beta)$ *for a state* $z\}$
7. $\rho(\alpha^*) = \bigcup_{n\in\mathbb{N}} \rho(\alpha^n)$ *where* $\alpha^{i+1} \hat{=} (\alpha;\alpha^i)$ *and* $\alpha^0 \hat{=} ?true$.

Definition 2 (Interpretation of dℒ formulas). *The interpretation* $\models$ *of a* dℒ *formula with respect to state* ν *is defined as follows.*

1. $\nu \models \theta_1 \sim \theta_2$ *iff* $[\![\theta_1]\!]_\nu \sim [\![\theta_2]\!]_\nu$ *for* $\sim \in \{=, \leq, <, \geq, >\}$
2. $\nu \models \phi \wedge \psi$ *iff* $\nu \models \phi$ *and* $\nu \models \psi$*, accordingly for* $\neg, \vee, \rightarrow, \leftrightarrow$
3. $\nu \models \forall x\, \phi$ *iff* $\omega \models \phi$ *for all* ω *that agree with* ν *except for the value of* x
4. $\nu \models \exists x\, \phi$ *iff* $\omega \models \phi$ *for some* ω *that agrees with* ν *except for the value of* x
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* $(\nu,\omega) \in \rho(\alpha)$

We write $\models \phi$ *to denote that* ϕ *is* valid*, i. e., that* $\nu \models \phi$ *for all* ν.

3.2 Refinement Relations

In order to justify our refactorings we introduce two refinement notions based on reachable states of hybrid programs.

Definition 3 (Projective Relational Refinement). *Let* $V \subseteq \Sigma$ *be a set of variables. Let* $|_V$ *denote the projection of relations or states to the variables in* V*. We say that hybrid program* α refines *hybrid program* γ w.r.t. the variables in V *(written as* $\alpha \sqsubseteq^V \gamma$*) iff* $\rho(\alpha)|_V \subseteq \rho(\gamma)|_V$*. If* $\alpha \sqsubseteq^V \gamma$ *and* $\gamma \sqsubseteq^V \alpha$ *then we speak of an* observability equivalence $\alpha \equiv^V \gamma$ w.r.t. V *between the systems.*

This notion of refinement guarantees that safety properties referring only to variables in V can be transferred from γ to α and liveness properties referring to V can be transferred conversely from α to γ. Projective relational refinement is monotonic w.r.t. hybrid program composition: if a hybrid program α refines a hybrid program β, i. e., $\alpha \sqsubseteq^V \beta$, then also $\alpha^* \sqsubseteq^V \beta^*$, $(\alpha \cup \gamma) \sqsubseteq^V (\beta \cup \gamma)$, $(\alpha;\gamma) \sqsubseteq^V (\beta;\gamma)$, and $(\gamma;\alpha) \sqsubseteq^V (\gamma;\beta)$ hold (cf. [16, App. A]).

If we want to exploit knowledge about the system parameters and reachable states when performing refinements we use a weaker notion of *partial refinement*. This allows us to show correctness of refactorings w.r.t. the assumptions and states that actually matter for a concrete original model (e. g., we may only care about those states that satisfy an invariant property of the original model).

Definition 4 (Partial Projective Relational Refinement). *We say that hybrid program* α partially refines *hybrid program* γ w.r.t. the variables in V and formula F *(written as* $\alpha \sqsubseteq_F^V \gamma$*) iff* $(?F;\alpha) \sqsubseteq^V (?F;\gamma)$. *If* $\alpha \sqsubseteq_F^V \gamma$ *and* $\gamma \sqsubseteq_F^V \alpha$ *we speak of a* partial observability equivalence $\alpha \equiv_F^V \gamma$ w.r.t. V and F.

Partial refinement still exhibits nice properties. Let $FV(\phi)$ denote the set of free variables of formula ϕ.

Lemma 1. *Let* α *and* γ *be two hybrid programs s.t.* $\alpha \sqsubseteq_F^V \gamma$. *Let* $\models G \to F$. *Assume* $\models G \to [\gamma]\psi$ *for some formula* ψ *with* $FV(\psi) \subseteq V$, *then* $\models G \to [\alpha]\psi$.

Proof. Assume $\alpha \sqsubseteq_F^V \gamma$, $\models G \to F$, $\models G \to [\gamma]\psi$ and $\not\models G \to [\alpha]\psi$ for some formula ψ with $FV(\psi) \subseteq V$. The semantics of the latter is that there is a state ν with $\nu \models G$ such that there is $(\nu,\omega) \in \rho(\alpha)$ with $\omega \not\models \psi$. Since we know that $G \to F$ is valid we also have that $\nu \models F$. Therefore, if $(\nu,\omega) \in \rho(\alpha)$ it also holds that $(\nu,\omega) \in \rho(?F;\alpha)$. However, since $\alpha \sqsubseteq_F^V \gamma$ and thus $(?F;\alpha) \sqsubseteq^V (?F;\gamma)$ we have that there is some ω' with $(\nu,\omega') \in \rho(?F;\gamma)$ s.t. $\omega|_V = \omega'|_V$. Furthermore, we have that $(\nu,\omega') \in \rho(\gamma)$. From $\models G \to [\gamma]\psi$ we can conclude that for all (ν,ω') we have that $\omega' \models \psi$. Since $\omega|_V = \omega'|_V$ and $FV(\psi) \subseteq V$ we have $\omega \models \psi$ by coincidence lemma [18, Lemma 2.6]. Thus, we conclude $\models G \to [\alpha]\psi$. □

Lemma 2. *Let* α *and* γ *be two hybrid programs s.t.* $\alpha \sqsubseteq_F^V \gamma$. *Let* $\models G \to F$. *Assume* $\models G \to \langle\alpha\rangle\psi$ *for some formula* ψ *with* $FV(\psi) \subseteq V$, *then* $\models G \to \langle\gamma\rangle\psi$.

Proof. The proof is analog to that of Lemma 1 (cf. [16, App. A]). □

We can derive two corollaries from these lemmas that cover the stronger properties of the total refinement relation (with $\alpha \sqsubseteq^V \gamma$ iff $\alpha \sqsubseteq_{\text{true}}^V \gamma$).

Corollary 1. *Let* α *and* γ *be two hybrid programs s.t.* $\alpha \sqsubseteq^V \gamma$. *If* $\models \phi \to [\gamma]\psi$ *for some formulas* ϕ *and* ψ *with* $FV(\psi) \subseteq V$, *then* $\models \phi \to [\alpha]\psi$.

Corollary 2. *Let* α *and* γ *be two hybrid programs s.t.* $\alpha \sqsubseteq^V \gamma$. *If* $\models \phi \to \langle\alpha\rangle\psi$ *for some formulas* ϕ *and* ψ *with* $FV(\psi) \subseteq V$, *then* $\models \phi \to \langle\gamma\rangle\psi$.

Relational refinement w.r.t. V can be logically characterized within $\mathsf{d}\mathcal{L}$. Subsequently, we use $\Upsilon_V \equiv \bigwedge_{v\in V} v = \tilde{v}$ to express a characterization of the V values in a state, where we always assume the variables $\tilde{v}$ to occur solely in Υ_V and nowhere else in any formula. Hence the formula $\langle\alpha\rangle\Upsilon_V$ identifies the states reachable by hybrid program α w.r.t. the variables in V. The variables in $\tilde{v}$ can then be used to recall this state; see [19] for details. In the following we use this notion to compare states reachable by one program with those reachable by another.

Theorem 1. *Let* α *and* γ *be hybrid programs. We have that* $\alpha \sqsubseteq^V \gamma$ *iff*

$$\models \big(\langle\alpha\rangle\Upsilon_V\big) \to \langle\gamma\rangle\Upsilon_V \tag{1}$$

Remark 1. Since the variables $\tilde{v}$ neither appear in α nor in γ we have that if $(\nu,\omega) \in (\rho(\alpha) \cup \rho(\gamma))$ then $[\![\tilde{v}]\!]_\nu = [\![\tilde{v}]\!]_\omega$ for all $\tilde{v}$ (cf. [18, Lemma 2.6]).

Proof. Let ν be arbitrary.

$\Rightarrow$ Assume that $\alpha \sqsubseteq^V \gamma$. Assume that $\nu \models \langle\alpha\rangle\Upsilon_V$ since otherwise there is nothing to show. This means that there is some state ω with $(\nu,\omega) \in \rho(\alpha)$ s.t. $\omega \models \Upsilon_V$. We fix any such ω arbitrarily. From $\alpha \sqsubseteq^V \gamma$ we know $\rho(\alpha)|_V \subseteq \rho(\gamma)|_V$. Using this and $(\nu,\omega) \in \rho(\alpha)$ there is ω' with $(\nu,\omega') \in \rho(\gamma)$ s.t. $[\![v]\!]_\omega = [\![v]\!]_{\omega'}$ for all $v \in V$. Furthermore, $[\![\tilde{v}]\!]_\omega = [\![\tilde{v}]\!]_{\omega'}$ for all $v \in V$ (Remark 1). Thus we conclude $\nu \models \langle\gamma\rangle\Upsilon_V$ by coincidence lemma [18, Lemma 2.6] since $FV(\langle\gamma\rangle\Upsilon_V) \subseteq V \cup \{\tilde{v} \mid v \in V\}$. Since ν was arbitrary we get (1) by the semantics of $\rightarrow$.

$\Leftarrow$ Assume (1). If $\rho(\alpha) = \emptyset$ then the proposition follows trivially. Otherwise consider any $(\nu,\omega) \in \rho(\alpha)$. Since no $\tilde{v}$ occurs in α, this implies $(\nu',\omega') \in \rho(\alpha)$ for $\nu'|_V = \nu|_V$, $\omega'|_V = \omega|_V$ and $[\![\tilde{v}]\!]_{\nu'} = [\![\tilde{v}]\!]_{\omega'} = [\![v]\!]_{\omega'}$. Thus, $\omega' \models \Upsilon_V$, so $\nu' \models \langle\alpha\rangle\Upsilon_V$. Therefore, (1) implies that there is ω'_γ with $(\nu',\omega'_\gamma) \in \rho(\gamma)$ s.t. $\omega'_\gamma \models \Upsilon_V$. Hence $[\![v]\!]_{\omega'_\gamma} = [\![\tilde{v}]\!]_{\omega'_\gamma} \overset{\text{Rem. 1}}{=} [\![\tilde{v}]\!]_{\nu'} \overset{\text{Rem. 1}}{=} [\![\tilde{v}]\!]_{\omega'} = [\![v]\!]_{\omega'}$ f.a. $v \in V$ because $\omega' \models \Upsilon_V$. Thus $\omega'|_V = \omega'_\gamma|_V$ and since, further, $(\nu',\omega'_\gamma) \in \rho(\gamma)$, $\nu|_V = \nu'|_V$ and $\omega|_V = \omega'|_V$ it follows that $(\nu|_V,\omega|_V) \in \rho(\gamma)|_V$. Since both ν and ω were arbitrary we get $\rho(\alpha)|_V \subseteq \rho(\gamma)|_V$ and conclude $\alpha \sqsubseteq^V \gamma$. □

In order to simplify refinement proofs we exploit prior knowledge about system trajectories. Suppose we want to establish refinement between two programs with loops, i. e., we want to show that $\alpha^* \sqsubseteq^V_F \gamma^*$; further assume that we have $\models F \rightarrow [\gamma^*]F$. We can use this knowledge to simplify a refinement proof.

Lemma 3. *For some set of variables V, let F be some formula with $FV(F) \subseteq V$. Under the assumption that $\models F \rightarrow [\gamma^*]F$ (in particular, F is an inductive invariant of γ^*) the following two statements are equivalent:*

$$\alpha^* \sqsubseteq^V_F \gamma^* \qquad (2) \qquad\qquad \models \big(\langle ?F;\alpha\rangle\Upsilon_V\big) \rightarrow \langle ?F;\gamma^*\rangle\Upsilon_V \qquad (3)$$

Observe that unlike in (1) we only need to argue about states reachable by exactly one execution of α in order to make a statement about α^*.

Proof. **(2) ⇒ (3)** If $\alpha^* \sqsubseteq^V_F \gamma^*$ then we know from Theorem 1 that (1) holds. Since $\rho(\alpha) \subseteq \rho(\alpha^*)$ we can conclude that (3) holds.

(3) ⇒ (2) Assume $\models F \rightarrow [\gamma^*]F$. Consider any $(\nu,\omega) \in \rho(?F;\alpha^*)$. To prove (2) we need to show that there is some $(\nu_\gamma,\omega_\gamma) \in \rho(?F;\gamma^*)$ with $\nu_\gamma|_V = \nu|_V$ and $\omega_\gamma|_V = \omega|_V$. If $\nu = \omega$ we are done, as $\rho(?F;\gamma^*)$ is reflexive from states where F holds by repeating 0 times. Otherwise, $?F;\alpha^*$ repeated at least once to get from ν to ω. Let μ s.t. $(\nu,\mu) \in \rho(?F;\alpha)$, $(\mu,\omega) \in \rho(\alpha^*)$. Let $\tilde{\nu},\tilde{\mu}$ s.t. $\tilde{\nu}|_V = \nu|_V$, $\tilde{\mu}|_V = \mu|_V$ and $[\![\tilde{v}]\!]_{\tilde{\nu}} = [\![\tilde{v}]\!]_{\tilde{\mu}} = [\![v]\!]_{\tilde{\mu}}$ f.a. $v \in V$, so $\tilde{\mu} \models \Upsilon_V$. The variables $\tilde{v}$ do not appear in $(?F;\alpha)$, so $(\tilde{\nu},\tilde{\mu}) \in \rho(?F;\alpha)$ still holds; thus, $\tilde{\nu} \models \langle ?F;\alpha\rangle\Upsilon_V$. Therefore, by (3) we have that $\tilde{\nu} \models \langle ?F;\gamma^*\rangle\Upsilon_V$. This means there is $(\tilde{\nu},\tilde{\mu}_\gamma) \in \rho(?F;\gamma^*)$ with $\tilde{\mu}_\gamma \models \Upsilon_V$. Observe that $[\![\tilde{v}]\!]_{\tilde{\mu}_\gamma} = [\![\tilde{v}]\!]_{\tilde{\mu}}$ since neither α nor γ change $\tilde{v}$ for any $v \in V$. There are only runs of this program if $\tilde{\nu} \models F$. Thus we conclude $\tilde{\mu}_\gamma \models F$ from $\models F \rightarrow [\gamma^*]F$. Furthermore, by $\tilde{\mu}_\gamma \models \Upsilon_V$ and $\tilde{\mu} \models \Upsilon_V$ we get $[\![v]\!]_{\tilde{\mu}_\gamma} = [\![\tilde{v}]\!]_{\tilde{\mu}_\gamma} = [\![\tilde{v}]\!]_{\tilde{\mu}} = [\![v]\!]_{\tilde{\mu}}$ f.a. $v \in V$. Thus, $\tilde{\mu} \models F$ by coincidence lemma [18, Lemma 2.6]. As $\tilde{\mu}|_V = \mu|_V$ and

$FV(F) \subseteq V$ we get $\mu \models F$ by coincidence lemma. As (ν, μ) was arbitrary this gives $\models F \to [\alpha]F$ and by soundness of induction $\models F \to [\alpha^*]F$. Hence $\{\omega \mid \nu \models F \text{ and } (\nu,\omega) \in \rho(\alpha)\} = \{\omega \mid \nu \models F \text{ and } (\nu,\omega) \in \rho(\alpha) \circ \rho(\alpha^*)\}$. However, this means by considering all $(\nu, \mu) \in \rho(?F; \alpha)$ we have constructed an argument for all elements of $\rho(?F; \alpha^*)$. Hence we get $\alpha^* \sqsubseteq_F^V \gamma^*$. □

3.3 Refactorings and Proof Obligations

We distinguish between structural and behavioral refactorings. *Structural refactorings* change the structure of a hybrid program without changing its reachable states. Structural refactorings ensure (partial) observability equivalence $\alpha \equiv_F^V \gamma$, which means that both safety and liveness properties can be transferred. *Behavioral refactorings* change a hybrid program in a way that also changes its behavior, i. e., the program reaches partly different states after the refactoring than before. Thus, behavioral refactorings need auxiliary gluing proofs (but not full reverification) to establish refinement relationships and transfer correctness properties from the original system γ to the refactored system α.

For transferring correctness properties we define the following proof obligations. Some of these obligations can be shown on a meta-level for all refactored α corresponding to γ; where this is impossible, the proofs have to be done for a particular refactoring instance $\gamma \rightsquigarrow \alpha$.

Observability equivalence proof. Observability equivalence ($\alpha \equiv^V \gamma$) is necessary to transfer safety and liveness properties referring to V at the same time. It can be characterized in $\mathsf{d}\mathcal{L}$ by $\models \big(\langle\alpha\rangle \Upsilon_V\big) \leftrightarrow \big(\langle\gamma\rangle \Upsilon_V\big)$. In addition we can use Lemma 3 in order to simplify reasoning for loops.

Safety relational refinement. Prove that all reachable states from the refactored model α are already reachable in the original model γ, i. e., for safety relational refinement use Theorem 1 to prove $\alpha \sqsubseteq^V \gamma$. In addition Lemma 3 simplifies loops.

Auxiliary safety proof. Prove that a refactored model α satisfies some safety properties under the assumption of an existing proof about the original model γ. The auxiliary safety proof patches this proof w.r.t. the changes made by the refactoring. Thus it is especially useful if neither observability equivalence nor relational refinement can be shown. Let $\forall^\gamma$ quantify universally over all variables that are changed in γ. The intuition is that, assuming $\models \forall^\gamma(\phi \to [\gamma]\phi)$ (i. e., ϕ is an inductive invariant of γ), we can close the identical parts in the proof from the assumption by axiom and only need to show correctness for the remaining, new parts of the refactored model. For auxiliary safety use an invariant of $\mathcal{I}(\phi) \equiv \big(\phi \wedge \forall^\gamma(\phi \to [\gamma]\phi)\big)$ for the refactored program α to prove

$$\big(F \wedge \mathcal{I}(\phi)\big) \to [\alpha^*]\psi \ . \tag{4}$$

Liveness relational refinement. To transfer liveness properties we have to prove the converse of the safety-preserving relational refinement proof, i. e., prove that all reachable states from the original model γ are also reachable in the refactored model α (use Theorem 1 to prove that $\gamma \sqsubseteq^V \alpha$).

Safety and liveness compliance/equivalence proof. Prove that a refactored model α satisfies the same property as its original model γ, i. e., for safety compliance prove $(\phi \to [\gamma]\psi) \to (\phi \to [\alpha]\psi)$, for safety equivalence prove $(\phi \to [\gamma]\psi) \leftrightarrow (\phi \to [\alpha]\psi)$. Liveness compliance/equivalence is analog to safety compliance/equivalence with $\langle\gamma\rangle$ in place of $[\gamma]$ and $\langle\alpha\rangle$ in place of $[\alpha]$. These are the generic fallback proof strategies that are always possible.

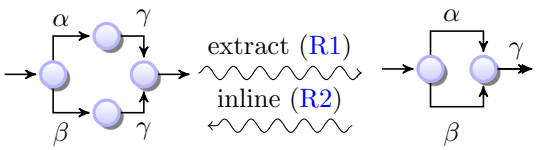
4 Structural Refactorings

Structural refactorings are observability-equivalent refactorings: they change the structure of a hybrid program without changing its reachable states. Structural refactorings are akin to the refactorings known from software engineering [8]. Here we discuss refactorings that arise specifically in hybrid system models; correctness proves can be found in [16, App. B].

We present refactorings as rewrite rules of the form $\frac{F}{\gamma \rightsquigarrow \alpha}$, meaning that program γ can be refactored into program α if preconditions F and the side conditions stated in footnotes are satisfied. We omit F and write $\gamma \rightsquigarrow \alpha$ if γ can be refactored into α unconditionally. We use $i \in I$ to denote the elements of some finite index set I when enumerating hybrid programs.

4.1 Extract Common Program

Duplicated (control) code is hard to maintain, because changes have to be made consistently at several places [8]. The *Extract Common Program* refactoring moves duplicated program fragments to a common path in the model. It can be used if the duplicated program parts are the last statements on a path.



Mechanics. Move the duplicated statements after the merging point of the paths.

(R1) $\bigcup_{i\in I}(\alpha_i;\gamma) \rightsquigarrow \left(\bigcup_{i\in I}\alpha_i\right);\gamma$ (R2) $\left(\bigcup_{i\in I}\alpha_i\right);\gamma \rightsquigarrow \bigcup_{i\in I}(\alpha_i;\gamma)$

Proof Obligations. None, because the original program and the refactored program are observability equivalent. Since set union distributes over relation composition, it is evident that $\rho\left(\bigcup_{i\in I}(\alpha_i;\gamma)\right)|_V = \rho\left(\left(\bigcup_{i\in I}\alpha_i\right);\gamma\right)|_V$. Thus, we can conclude that $\bigcup_{i\in I}(\alpha_i;\gamma) \equiv^V \left(\bigcup_{i\in I}\alpha_i\right);\gamma$.

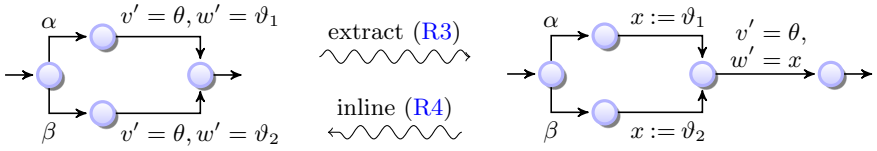
Variation: Inline Program. Duplicate γ into each branch.

4.2 Extract Continuous Dynamics

Scattered continuous dynamics on multiple paths in a hybrid program make it hard to introduce explicit computation delay into models of CPS [11], because those would need to be duplicated in any of the respective paths as well. The *Extract Continuous Dynamics* refactoring collects the continuous dynamics from

multiple paths and introduces a unified differential equation after the merge point of those paths. The continuous dynamics of those paths have to be the final statements on their respective path. If the original paths encode deviating control actions, new variables are introduced to maintain different control actions on different paths.

Mechanics. Move the continuous dynamics after the merging point of the paths; capture path differences in new variables and set their values accordingly.



$$\text{(R3)}\ \frac{\forall v \in V(\theta) \cup \bigcup_{i\in I} V(\vartheta_i).\ v \notin BV(\mathcal{D}(\theta)) \cup \bigcup_{i\in I} BV(\mathcal{D}(\vartheta_i))}{\bigcup_{i\in I}\big(\alpha_i; (v' = \theta, w' = \vartheta_i)\big) \ \rightsquigarrow\ \big(\bigcup_{i\in I}(\alpha_i; x := \vartheta_i)\big); (v' = \theta, w' = x)\big)}\ [1]$$

$$\text{(R4)}\ \big(\bigcup_{i\in I}(\alpha_i; x := \vartheta_i)\big); (v' = \theta, w' = x)\big) \ \rightsquigarrow\ \bigcup_{i\in I}\big(\alpha_i; (v' = \theta, w' = \vartheta_i)\big)$$

[1] with fresh variable $x \notin V$; we denote by $\mathcal{D}(\theta)$ a differential equation system containing the term θ; we refer to the variables in θ, by $V(\theta)$; let $BV(\mathcal{D}(\cdot))$ denote the variables changed by the ODE $\mathcal{D}(\cdot)$.

Proof Obligations. None, because the original program and the refactored program are observability equivalent, i. e.,

$$\big(\bigcup_{i\in I}(\alpha_i; x := \vartheta_i)\big); (v' = \theta, w' = x)\big) \equiv^V \bigcup_{i\in I}\big(\alpha_i; (v' = \theta, w' = \vartheta_i)\big)\ .$$

Let $\mathcal{D}(\theta)$ be a differential equation system (with or without evolution domain constraint) containing the term θ. For some fresh variable $x \notin V$ we have that $\rho(\mathcal{D}(\theta))|_V = \rho(x := \theta; \mathcal{D}(x))|_V$ under the condition that no variable that occurs in θ has a derivative in $\mathcal{D}(\theta)$. If we perform this operation on all branches, we can use distributivity of sequential composition over choice (i. e., the Extract Common Program refactoring) to move the common part into a single statement.

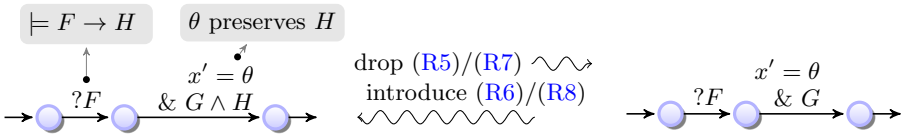
Variation: Inline Continuous Dynamics. Duplicate the continuous dynamics into each path and push the path variable assignments into the continuous dynamics.

4.3 Drop Implied Evolution Domain Constraint

The *Drop Implied Evolution Domain Constraint* refactoring removes those constraints from the evolution domain that are already guaranteed by the discrete controller. It reduces constraint duplication and is also useful as an intermediate step in composite refactorings (see Section 6.2 for an example). The refactoring can be used when the context specifies constraints that are at least as strong as the evolution domain constraints (e. g., in the inductive invariant or in a test

prior to the continuous dynamics), and both discrete control and continuous dynamics only change the relevant fragment of the context in a way that preserves the evolution domain constraints.

Mechanics. Drop the implied constraints from the evolution domain.



$$\text{(R5)}\ \frac{F \to H \qquad F \to [x' = \theta \,\&\, G]H}{?F; x' = \theta \,\&\, G \wedge H \ \rightsquigarrow\ ?F; x' = \theta \,\&\, G} \qquad \text{(R6)}\ ?F; x' = \theta \,\&\, G \rightsquigarrow ?F; x' = \theta \,\&\, G \wedge H$$

$$\text{(R7)}\ ?F; x' = \theta \,\&\, G \wedge H \rightsquigarrow ?F; x' = \theta \,\&\, G \qquad \text{(R8)}\ \frac{F \to H \qquad F \to [x' = \theta \,\&\, G]H}{?F; x' = \theta \,\&\, G \ \rightsquigarrow\ ?F; x' = \theta \,\&\, G \wedge H}$$

Liveness Proof Obligations (R7). None, because projective partial refinement, i. e., $\left(?F; x' = \theta \,\&\, G \wedge H\right) \sqsubseteq_F^V \left(?F; x' = \theta \,\&\, G\right)$ holds.

Safety Proof Obligations (R5). We have to show that H is a differential invariant [21] (which, with $\models F \to H$, implies H can be dropped).

Variation: Introduce Evolution Domain Constraint. Safety properties transfer directly (R6), because the refactored program is a partial refinement of the original program, i. e., $\left(?F; x' = \theta \,\&\, G \wedge H\right) \sqsubseteq_F^V \left(?F; x' = \theta \,\&\, G\right)$ holds. For liveness (R8), prove that H is a differential invariant (which, with $\models F \to H$, implies H can be introduced).

5 Behavioral Refactorings

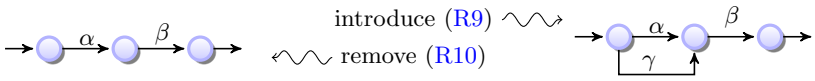
Behavioral refactorings change the states reachable by a hybrid system. This means that the proofs for the original model and those for a refactored model need auxiliary gluing proofs to transfer correctness properties. In most cases, these auxiliary proofs can reuse significant parts of the original proof. Correctness proofs that derive proof obligations can be found in [16, App. C].

5.1 Introduce Control Path

The initial models of a system are often simplified in order to reduce time-to-market or manage verification complexity. These initial models are later refined with more sophisticated control options (e. g., have multiple braking variants) once the initial model is provably safe. The *Introduce Control Path* refactoring introduces a new control path and adds it as a non-deterministic choice to the existing paths. The new control path must preserve the original invariant.

Mechanics. Introduce a new path via nondeterministic choice to existing paths.

(R9) $\alpha;\beta \ \rightsquigarrow\ (\alpha \cup \gamma);\beta$ (R10) $(\alpha \cup \gamma);\beta \ \rightsquigarrow\ \alpha;\beta$



Liveness Proof Obligations. None, because we get $(\alpha;\beta) \sqsubseteq^V \big((\alpha \cup \gamma);\beta\big)$ from the definition of the refinement relation.

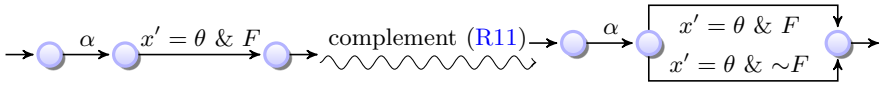
Safety Proof Obligations. Use an auxiliary safety proof (4). An example of such a proof is given in Section 6.1.

Variation: Remove Control Path. Again, we get $(\alpha;\beta) \sqsubseteq^V \big((\alpha \cup \gamma);\beta\big)$ from the definition of our refinement relation. Thus, safety properties can be transferred from the original model $(\alpha \cup \gamma);\beta$ to the refactored model $\alpha;\beta$ unconditionally.

5.2 Introduce Complementary Continuous Dynamics

Non-exhaustive evolution domain constraints are used to restrict differential equations to realistic regions (e. g., model braking as negative acceleration inside the region of positive velocities). But if misused, reasonable real-world behavior is sometimes excluded from a model. Such a model can be proven correct, but only because the evolution domain constraints limit unsafe behavior to stay within the safe states. *Introduce Complementary Continuous Dynamics* introduces a copy of the original differential equations with weak negation (i. e., negation retaining boundaries, denoted by $\sim F$) of the original evolution domain constraints. This way, an event is still reliably detected while the refactoring ensures that no behavior is excluded from the model. It is then the responsibility of the discrete controller to keep the system inside the safe region. The refactoring ensures that instantaneous reactions in the event detection part of the evolution domain does not clip reasonable behavior just for the sake of detecting an event.

Mechanics. Introduce a nondeterministic choice to a copy of the continuous dynamics, which uses the weak negation of the event detection evolution domain constraints of the original model. Merge after the continuous dynamics.



(R11) $\alpha; x' = \theta \,\&\, F \;\rightsquigarrow\; \alpha; (x' = \theta \,\&\, F \cup x' = \theta \,\&\, \sim F)$

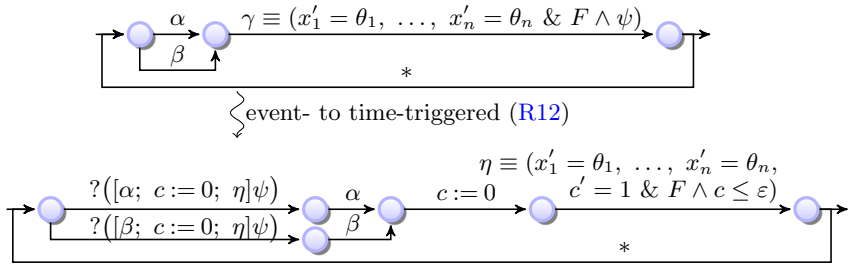
Liveness Proof Obligations. None, because from the transition semantics of $\mathsf{d}\mathcal{L}$ we get $\rho\big(\alpha; (x' = \theta \,\&\, F \cup x' = \theta \,\&\, \sim F)\big)|_V = \rho\big((\alpha; x' = \theta \,\&\, F) \cup (\alpha; x' = \theta \,\&\, \sim F)\big)|_V$ since set union distributes over relation composition. With $\rho(\alpha; x' = \theta \,\&\, F)|_V \subseteq \rho\big((\alpha; x' = \theta \,\&\, F) \cup (\alpha; x' = \theta \,\&\, \sim F)\big)|_V$ we conclude that $\big(\alpha; x' = \theta \,\&\, F\big) \sqsubseteq^V \big(\alpha; (x' = \theta \,\&\, F \cup x' = \theta \,\&\, \sim F)\big)$ holds, i. e., the refactored model is a liveness relational refinement of the original model.

Safety Proof Obligations. For safety, show that the controller with subsequent complementary dynamics only reaches states that are already reachable with the original dynamics, i. e., show $\big(\langle \alpha; x' = \theta \,\&\, \sim F \rangle \Upsilon_V\big) \rightarrow \big(\langle \alpha; x' = \theta \,\&\, F \rangle \Upsilon_V\big)$.

5.3 Event- to Time-Triggered Architecture

Event-triggered architecture [10] is often easier to verify than time-triggered architecture, because the burden of detecting critical events is not on the controller but encoded as evolution domain constraint in the continuous dynamics. Hence, reactions are assumed to work instantaneous, which, however, makes event-triggered architecture hard if not impossible to implement. In a time-triggered architecture, in contrast, the controller samples the environment at regular time intervals. This introduces additional delay in event detection, which must be accounted for in the safety constraints of the controller. The *Event- to Time-Triggered Architecture* refactoring (suggested in [12]) turns a hybrid program with event-triggered architecture into one using time-triggered architecture.

Mechanics. Introduce a clock variable (e. g., c) with constant slope and an upper bound for the clock (e. g., ε) as evolution domain constraint. Reset the clock to 0 before the continuous dynamics are executed. Move the original evolution domain constraint as a test before the continuous dynamics. Strengthen the test such that it can keep the system safe under the current control decision.



(R12) $\left(\bigcup_{i\in I}\alpha_i\right);(x'=\theta\,\&\,F\wedge\psi)\rightsquigarrow\left(\bigcup_{i\in I}(?[\alpha_i;c:=0;\eta]\psi;\alpha_i)\right);c:=0;\eta$ [1]

[1] $\eta\equiv x_1'=\theta_1,\ldots,x_n'=\theta_n,c'=1\,\&\,F\wedge c\leq\varepsilon$ and fresh variables $c,\varepsilon\notin V$ with $\varepsilon>0$

Safety Proof Obligations. Show an auxiliary safety proof (4). Note, that in the original model the evolution domain constraint ψ holds throughout γ. In the refactored model, the tests $?\big([\alpha;c:=0;\eta]\psi\big)$ and $?\big([\beta;c:=0;\eta]\psi\big)$ check that ψ will hold for up to duration ε before α respectively β are executed. Therefore, ψ holds throughout η, because η contains the evolution domain constraint $c\leq\varepsilon$.

Observe that the test introduced in this refactoring uses a modality in order to exactly characterize the states for which the specific control action ensures safety during the next control cycle. This corresponds to model-predictive control [20]. Further refactorings can be used to replace this test by either an equivalent first-order formula (usually for some cases even *true*) or if impossible (or impractical) a stronger first-order formula. See for example [23] how to discover such formulas.

6 Safe Refactoring Examples with Refinement Reasoning

In this section we exemplify how to satisfy the proof obligations set forth by our refactorings. We use a simple hybrid model of a car inspired by [14]. The car has

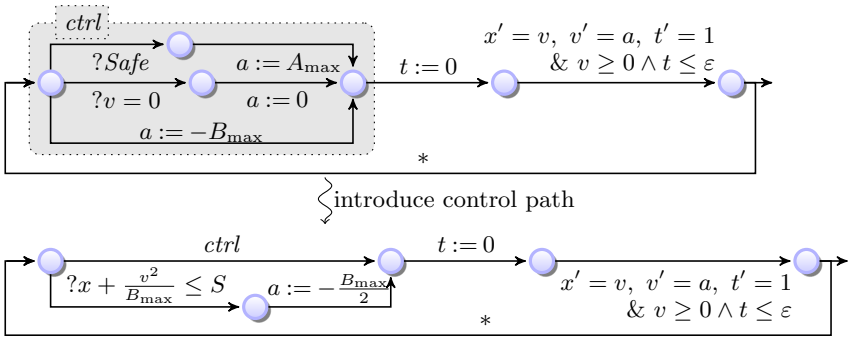


Fig. 1: Example of the effect of the Introduce Control Path refactoring.

three control choices: (i) it can accelerate with maximum acceleration $a := A_{\max}$ if it is safe to do so (indicated by safety property *Safe*), (ii) it can remain stopped by $a := 0$ if it is already stopped ($?v = 0$), and (iii) it can unconditionally brake with maximum braking force $a := -B_{\max}$. Its driving dynamics are modeled using the ideal-world differential equation system $x' = v,\ v' = a\ \&\ v \geq 0$.

6.1 Introduce Control Path

Let us assume we proved $\phi \rightarrow [car^*]\psi$ for some ϕ, ψ and we want the same safety guarantees about a refactored model $\widetilde{car}$ with an additional control path for moderate braking, i. e., we want to show that $\phi \rightarrow [\widetilde{car}^*]\psi$. Fig. 1 depicts the original model and the refactored model as state transition systems.

To reduce the proof effort for the refactored model we exploit the systematic way in which the Introduce Control Path refactoring changes the original model to produce the refactored model: the refactoring introduces a new branch without touching the remainder of the model. We can do an auxiliary safety proof that leverages the fact that we have a safety proof about the original model in those branches that are still present in the refactored model. As an inductive invariant for $\widetilde{car}$, we use $\mathcal{I}(\phi) \equiv \phi \wedge \forall x \forall v\, (\phi \rightarrow [car]\phi)$, which is the original invariant ϕ strengthened with the assumption that we have a proof about the original model. We thus prove formula (4), i. e., $\mathcal{I}(\phi) \rightarrow [\widetilde{car}^*]\psi$, see Sequent Proof 1.

6.2 Event- to Time-Triggered Architecture

The event- to time-triggered architecture refactoring is a composite refactoring that radically shifts the control paradigm between the original and the refactored model. Still, if some branches of the original model are retained in the refactored model we can reduce the overall verification effort with an auxiliary safety proof. Fig. 2 illustrates the refactoring operations.

First, *inline program* and *drop implied evolution domain constraint* transform the original model into an intermediate form with branches for braking and remaining stopped being symbolically equivalent to those in the original model.

Proof 1 Proof sketch of an auxiliary safety proof for *Introduce Control Path*. We use the abbreviations $A \equiv ?Safe; a := A_{\max}$, $C \equiv ?v = 0; a := 0$, $B \equiv a := -B_{\max}$, $B_r \equiv ?x + \frac{v^2}{B_{\max}} \leq S; a := -\frac{B_{\max}}{2}$, and $P \equiv x' = v, v' = a \;\&\; v \geq 0$.

$$(\wedge\text{r})\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \qquad (\wedge\text{l})\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \qquad (\forall\text{l})\ \frac{\Gamma, \phi(\theta) \vdash \Delta}{\Gamma, \forall x\, \phi(x) \vdash \Delta}$$

$$([]\ \text{gen})\ \frac{\Gamma \vdash [\alpha]\phi, \Delta \quad \phi \vdash \psi}{\Gamma \vdash [\alpha]\psi, \Delta} \qquad (ax)\ \frac{}{\Gamma, \phi \vdash \phi, \Delta} \qquad (\text{Wl})\ \frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\phi, [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \vdash [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi}\,ax \quad \dfrac{\dfrac{}{\phi \vdash [\mathrm{B}_r][P]\phi}\,\text{prove new branch}}{\phi, [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \vdash [\mathrm{B}_r][P]\phi}\,\text{Wl}}{\phi, [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \vdash [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \wedge [\mathrm{B}_r][P]\phi}\,\wedge\text{r}}{\phi, [(\mathrm{A}\cup\mathrm{C}\cup\mathrm{B});\ P]\phi \vdash [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \wedge [\mathrm{B}_r][P]\phi}\,[;]}{\phi, \forall x \forall v\,(\phi \to [(\mathrm{A}\cup\mathrm{C}\cup\mathrm{B});\ P]\phi) \vdash [\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}][P]\phi \wedge [\mathrm{B}_r][P]\phi}\,\forall\text{l},\forall\text{l}}{\phi, \forall x \forall v\,(\phi \to [(\mathrm{A}\cup\mathrm{C}\cup\mathrm{B});\ P]\phi) \vdash [(\mathrm{A}\cup\mathrm{C}\cup\mathrm{B}\cup\mathrm{B}_r);\ P]\phi}\,[;],[\cup]}{\phi, \forall x \forall v\,(\phi \to [car]\phi) \vdash [\widetilde{car}]\phi}\,\text{expand}$$

$$\dfrac{\dfrac{\dfrac{\dots \quad \dfrac{}{\phi, \forall x \forall v(\phi \to [car]\phi) \vdash \phi \wedge \forall x \forall v(\phi \to [car]\phi)}\,ax}{\phi, \forall x \forall v(\phi \to [car]\phi) \vdash [\widetilde{car}](\phi \wedge \forall x \forall v(\phi \to [car]\phi))}\,[]\ \text{gen}}{\phi \wedge \forall x \forall v(\phi \to [car]\phi) \vdash [\widetilde{car}](\phi \wedge \forall x \forall v(\phi \to [car]\phi))}\,\wedge\text{l}}{\mathcal{I}(\phi) \vdash [\widetilde{car}]\mathcal{I}(\phi)}\,\text{expand}$$

$$\dfrac{\dfrac{\dfrac{}{\mathcal{I}(\phi) \vdash \mathcal{I}(\phi)}\,ax \quad \dots \quad \dfrac{\dfrac{}{\phi \vdash \psi}}{\phi \wedge \forall x \forall v\,(\phi \to [car]\phi) \vdash \psi}\,\wedge\text{l},\text{Wl}}{\mathcal{I}(\phi) \vdash \psi}\,\text{expand}}{\mathcal{I}(\phi) \vdash [\widetilde{car}^*]\psi}\,[*]$$

Then, we introduce the time-triggered acceleration decision and use an auxiliary safety proof to show that the changed acceleration branch ensures the inductive invariant (the mechanics of this proof are similar to those in Sequent Proof 1). Finally, *differential auxiliary* [21] introduces t with initial value $t := 0$ and differential equation $t' = 1$ into the braking and remaining stopped branches, and *extract common program* transforms into the final refactored form.

7 Conclusion

We introduced proof-aware refactoring operations for hybrid systems. The notion of projective relational refinement allows us to make strong correctness statements about some refactorings on a meta-level for all hybrid programs. Where this is impossible, our refactoring operations construct proof obligations for showing that the resulting refactored model is a correct refinement of the particular original model instance.

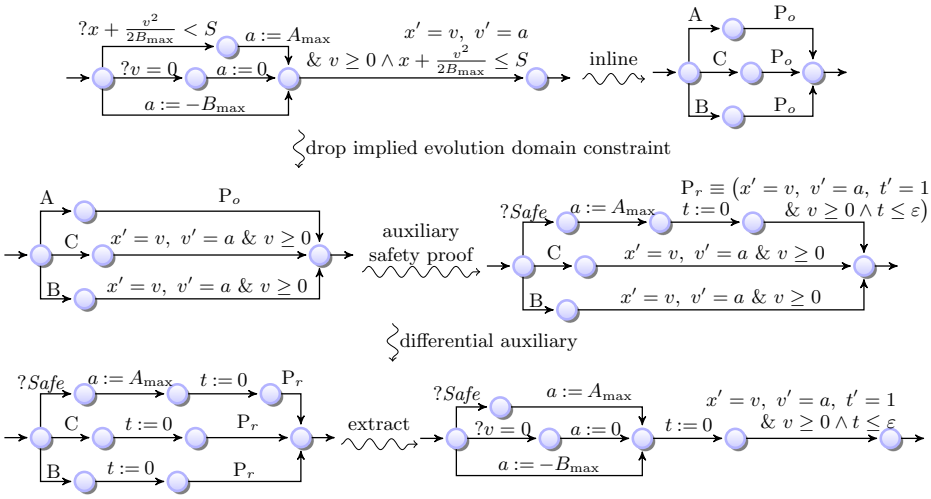


Fig. 2: Intermediate steps in event- to time-triggered architecture refactoring.

We are in the process of implementing the refactoring operations in our verification-driven engineering tool Sphinx [15]. Future work includes building a catalog of structural and behavioral refactorings and the evaluation of its refactoring operations with case studies in hybrid system verification. We plan to further extend the constructed proof obligations, for example with auxiliary liveness proofs to patch existing liveness proofs when liveness relational refinement cannot be shown. Another interesting direction for research is to develop additional refinement notions based on hybrid games [24] for transferring liveness properties about models with sensor uncertainty and actuator disturbance.

Acknowledgments. We want to thank the anonymous reviewers for their valuable feedback. This material is partially supported by the NSF under grants CNS-1054246, CNS-0926181, CNS-1035800, CNS-0931985, by DARPA under FA8750-12-2-0291, by the US DOT under # DTRT12GUTC11, and by Austrian BMVIT grants FFG 829598, FFG 838526, FFG 838181. The research leading to these results has received funding from the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement n° PIOF-GA-2012-328378.

References

1. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6) (2010)
2. Alur, R.: Can we verify cyber-physical systems?: technical perspective. Commun. ACM 56(10), 96 (2013)
3. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional modeling and refinement for hierarchical hybrid systems. J. Log. Algebr. Program. 68(1-2), 105–128 (2006)

4. Börger, E.: The ASM refinement method. Formal Aspects of Computing 15(2-3), 237–257 (2003)
5. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B.H., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. 14(4), 583–604 (2003)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
7. Doyen, L., Henzinger, T.A., Raskin, J.F.: Automatic rectangular refinement of affine hybrid systems. In: Pettersson, P., Yi, W. (eds.) FORMATS. LNCS, vol. 3829, pp. 144–161. Springer (2005)
8. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring—Improving the Design of Existing Code. Addison-Wesley (1999)
9. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
10. Kopetz, H.: Event-triggered versus time-triggered real-time systems. In: Karshmer, A.I., Nehmer, J. (eds.) Operating Systems of the 90s and Beyond. LNCS (1991)
11. Kouskoulas, Y., Platzer, A., Kazanzides, P.: Formal methods for robotic system control software. Tech. Rep. 2, Johns Hopkins University APL (2013)
12. Kouskoulas, Y., Renshaw, D., Platzer, A., Kazanzides, P.: Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In: Belta, C., Ivancic, F. (eds.) HSCC. ACM (2013)
13. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Robotics: Science and Systems (2013)
14. Mitsch, S., Loos, S.M., Platzer, A.: Towards formal verification of freeway traffic control. In: Lu, C. (ed.) ICCPS. pp. 171–180. IEEE (2012)
15. Mitsch, S., Passmore, G.O., Platzer, A.: A vision of collaborative verification-driven engineering of hybrid systems. In: Kerber, M., Lange, C., Rowat, C. (eds.) DoForm. pp. 8–17. AISB (2013)
16. Mitsch, S., Quesel, J.D., Platzer, A.: Refactoring, refinement, and reasoning: A logical characterization for hybrid systems. Tech. Rep. CMU-CS-14-103, Carnegie Mellon (2014)
17. Opdyke, W.F.: Refactoring Object-oriented Frameworks. Ph.D. thesis, Champaign, IL, USA (1992), uMI Order No. GAX93-05645
18. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
19. Platzer, A.: A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. Logical Methods in Computer Science 8(4), 1–44 (2012), special issue for selected papers from CSL'10
20. Platzer, A.: Logics of dynamical systems. In: LICS. pp. 13–24. IEEE (2012)
21. Platzer, A.: The structure of differential invariants and differential cut elimination. Logical Methods in Computer Science 8(4), 1–38 (2012)
22. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS (2008)
23. Platzer, A., Quesel, J.D.: European Train Control System: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. LNCS, Springer (2009)
24. Quesel, J.D., Platzer, A.: Playing hybrid games with KeYmaera. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR. LNCS, vol. 7364, pp. 439–453. Springer (2012)
25. Schneider, S., Treharne, H., Wehrheim, H.: The behavioural semantics of Event-B refinement. Formal Aspects of Computing pp. 1–30 (2012)
26. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)