

Refactoring-as-Propositions: Proved Refactoring of Hybrid Systems via Proved Refinements

Enguerrand Prebet(✉) and André Platzer

Karlsruhe Institute of Technology, Karlsruhe, Germany
{enguerrand.prebet,platzer}@kit.edu

Abstract. Cyber-physical systems are inherently complex due to their connection between software and the physical world. Iterative design reduces their complexity, but increases the need to repeatedly recheck their safety in full after every change. We introduce the *refactoring-as-propositions* principle in which refactorings are represented as propositions along with a method for proving that system refactorings preserve their required properties by transferring the proof along the respective modification. It is based on *differential refinement logic* (dRL), with which one can simultaneously and rigorously refer to properties of the systems and the relation between a refactored system and its original version. Refinements represent a uniform way of expressing different types of hybrid system refactorings, including those that introduce auxiliary variables. Furthermore, we show how these refactorings can be proved automatically, and/or reduce to a modular proof solely about the local change rather than about the whole system.

Keywords: Refactoring · Differential dynamic logic · Refinement · Hybrid systems

1 Introduction

Cyber-physical systems (CPS) are crucial but subtle to get right. Their design needs to deal with several interacting features, making it challenging to get them all right all at the same time. That is where iterative designs help when starting with simpler systems, figuring them out, and then iteratively adding more and more complicated features. While this reduces the effort per design, the cost is that systems need to be analyzed for safety repeatedly, often with similar but different arguments. CPS refactoring techniques [17] offer specific refactoring operations on CPS that preserve safety when only certain differences are reproved. But the correctness of the safety verification of the final system depends on the correct implementation and application of every refactoring operation.

This paper shows that all of those prior CPS refactoring operations can, instead, be derived syntactically as propositions in *differential refinement logic* (dRL) [15,23]. Instead of verifying refactoring algorithms for specific properties,

© The Author(s) 2026
A. Biere et al. (Eds.): IJCAR 2026, LNAI 16689, pp. 324–341, 2026.
https://doi.org/10.1007/978-3-032-32592-1_19

refactorings-as-propositions are refinements proved in dRL, and can be reused independently of the systems and properties considered. The correctness of such refactoring applications then boils down to dRL's refinements-to-properties axiom [≤]. Since dRL has a sound proof calculus [15], syntactic derivations in dRL give a sound implementation of CPS refactoring techniques. Furthermore, dRL's proof calculus is a uniform substitution calculus [23], so that the resulting derivations can be implemented in the prover KeYmaera X solely on top of its small soundness-critical prover microkernel. This paves the way to provably correct automatic CPS refactoring technique implementations.

While most standard refactorings are usually meant to preserve the same semantics, it gets subtle for refactorings that introduce fresh auxiliary variables in imperative programs, which dRL can model. Consider adding the program $?x = 0; x := 1$ in a system where x is fresh. This snippet first tests if the value of x is 0 and aborts execution otherwise before setting the value of x to 1. If neither the rest of the system nor the postcondition of interest use x, the corresponding execution should be unaffected, and so should be the validity of the postcondition. Except this reasoning *only* holds if the snippet is not added within a loop. In a loop, the second iteration of the loop would abort since the test $x = 0$ would fail now that x has been set to 1. This drastically alters the behavior despite not interacting with any existing variable! Conversely, an assignment $x := \theta$ where θ is any polynomial expression can safely be added no matter what variables occur in θ. Having a formal proof for handling correct refactorings, based on the notion of *ghost refinement*, gives a general understanding when such subtle changes are sound, while remaining automatable.

Contributions. We introduce the refactoring-as-propositions principle by providing techniques for transforming proved refinements in dRL to generic, proved refactoring techniques for hybrid programs. We do so for three classes of refinements: globally sound refinements, locally sound refinements (or conditional refinements), and ghost refinements, all of which require negligible changes to dRL's axiomatization. It allows for an implementation of these techniques in the theorem prover KeYmaera X [11], enabling sound automatic refactorings via refinement, while keeping the soundness-critical kernel small. We show how these different classes of refactorings can be applied in case studies.

Related Works. Refactorings for differential dynamic logic (dL) [18], which dRL is an extension of, has been studied by adding a separate layer on top of the logic [17]. Other works include compositions [16] and parallelism [6], which decompose a hybrid system into parts. Thus, if a component is updated, then only part of the proof needs to be updated if it still satisfies the same contract. The Event-B method [1] initially designed for discrete programs has been extended to continuous evolutions [5,9]. The base logic of Event-B is not expressive enough for modelling continuous evolution so it must rely on theories [7]. Hybrid Event-B [3,4] builds upon it to handle hybrid programs. It can be used with the Rodin tool [2,26] to generate the proof obligations when proving refinements. A detailed comparison between Event-B and dL is provided in the literature [8].

2 Background: Differential Refinement Logic

This section provides an overview of differential refinement logic (dRL) [12,21,23]. After recalling its syntax and semantics, we briefly present how the logic can be used for proofs, both in theory and using the theorem prover KeYmaera X [11].

2.1 Syntax

At its core, dRL is a first-order multi-modal logic where the modalities are interpreted by hybrid programs, and with a dedicated refinement operator. dRL terms evaluate over real states, that is, valuations from variables to values in $\mathbb{R}$.

Definition 1 (Term). Terms *are defined by the following grammar where* θ, η *are terms,* x *is variable, and* n *is an integer.*

$$\theta, \eta ::= x \mid n \mid \theta + \eta \mid \theta \cdot \eta \mid (\theta)'$$

Beyond standard arithmetic, terms can also have differentials $(\theta)'$. This is to account for the continuous fragment of hybrid programs. For that purpose, each variable x is associated a differential variable x'. Their behaviors are linked during differential equations, and differential variables are essential for the soundness of axioms like DI and ODE.

Definition 2 (Formula). Formulas *are defined by the following grammar where* θ, η *are terms,* ϕ, ψ *are formulas and* α, β *are hybrid programs (Definition 3).*

$$\phi, \psi ::= \theta \leq \eta \mid \neg\phi \mid \phi \wedge \psi \mid \forall x\, \phi \mid [\alpha]\phi \mid \alpha \leq \beta$$

dRL formulas may be true on only a subset of states. We focus on the additional constructs not in first-order logic which are the modalities and refinements. A formula $[\alpha]\phi$ expresses that the formula ϕ always holds after executing the hybrid program α. Thus the formula $\phi \rightarrow [\alpha]\psi$ can be understood as the standard Hoare triple $\{\phi\}\alpha\{\psi\}$ but for hybrid systems. By duality, there is another modality $\langle\alpha\rangle\phi \;:=\; \neg[\alpha]\neg\phi$ that holds in a state s, if there exists an execution of α starting from s whose final state satisfies ϕ.

Refinements $\alpha \leq \beta$ on the other hand relate the output states of α, not to a formula ϕ but to the output states of another program β. A refinement $\alpha \leq \beta$ holds in a state s, if all states reachable by executing α from s are also reachable by some execution of β from s. Program equivalence, written $\alpha = \beta$, simply denotes the symmetric version of refinement, and can be seen as syntactic sugar for $\alpha \leq \beta \wedge \beta \leq \alpha$. Having refinements as a construct directly in the logic allows to reason and prove partial refinements, i.e., refinements that only holds under some assumption or after some execution like $[\alpha](\beta \leq \gamma)$. This idea is central for the refactoring techniques of Sect. 3.2.

Definition 3 (Hybrid Program). Hybrid program *are defined by the following grammar where* θ *are terms,* ψ *are formulas and* α, β *are hybrid programs.*

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

Hybrid programs form a Kleene Algebra with tests [14] and represent relations from input states to output states. The test $?\phi$ behaves as a noop in states where ϕ holds and aborts otherwise. The assignment $x := \theta$ updates the value of x in the current state to the computed value of θ. Similarly, the nondeterministic assignment $x := *$ updates the value of x to any possible real value. Practically, nondeterministic assignment are usually followed by a test to limit the possible values, e.g., $x := *; ?(0 \leq x \wedge x \leq 10)$ updates x to be any real inside the interval $[0, 10]$. Nondeterministic choice $\alpha \cup \beta$ can behave as α or as β. Sequence composition $\alpha; \beta$ behaves as α first and then proceeds as β. The *differential equation* $x' = \theta \,\&\, \psi$ behaves like a continuous evolution where both the differential equation $x' = \theta$ and the evolution domain ψ holds and can be extended for multiple ODEs. During the evolution, both the value of the variables x and x' are modified. The duration of the evolution is nondeterministic: it may stop immediately, in which case only x' is modified, or continue up until ψ is about to be violated, etc. The evolution domain ψ can be used to model physical reality – e.g., a ball stays above the floor – or internal triggers by stopping the evolution so that another part of the system is executed before continuing. Finally, the Kleene star α^* executes α a nondeterministic number of times, possibly none.

The highly nondeterministic behavior of hybrid programs pairs well with the box modality as it ensures the postcondition for all possible executions. More concrete, deterministic, programs can then be proved safe by refining the nondeterministic one. Another benefit of nondeterminism is regarding differential equations. By definition $[x' = \theta \,\&\, \psi]\phi$ means that ϕ only holds in the final state. However, since the duration can be arbitrary, proving such formula amounts to proving that ϕ holds throughout the whole continuous evolution.

2.2 Calculus

dRL comes with an axiomatization that is used to prove validity of its formulas. Given that formulas may hold only on subsets of states, we say that a formula is *valid* if it holds for all states. Similarly, a proof rule is sound if the validity of its premises implies the validity of its conclusion.

In Fig. 1, we present a subset of axioms and rules of dRL [23] to highlight its expressiveness and reasoning capabilities. If the differential of a term is 0 during continuous evolution, it is a differential invariant, and the value of the term is determined by its initial value (DI). Axioms [;] and $;_{\leq}$ decompose a proof for a modality and refinement into a proof for their respective constituant. Axiom $[\leq]$ says that if all outputs of a program β satisfy a formula ϕ, then all outputs of any refined program α also do. Refinement is reflexive ($\leq_r$) and transitive ($\leq_t$). Axiom ? relates a refinement between tests and their respective formula. A refinement between differential equations only holds if the equation of the more general holds throughout the execution of the more refined program (ODE). When the domain does not change, this gives a program equivalence instead. The basic usage of the axioms and rules is to use propositional reasoning to reduce the expression in blue by a simpler one, where the hybrid programs are decomposed in simpler subprograms.

$$([:=])\ [x := \theta]\phi(x) \leftrightarrow \phi(\theta)$$

$$(DI)\ \big([x' = \theta]\,\eta = 0 \leftrightarrow \eta = 0\big) \leftarrow [x' = \theta]\,(\eta)' = 0$$

$$([\leq])\ \alpha \leq \beta \rightarrow ([\beta]\phi \rightarrow [\alpha]\phi)$$

$$(\leq_t)\ \alpha \leq \beta \rightarrow (\beta \leq \gamma \rightarrow \alpha \leq \gamma)$$

$$(;_\leq)\ \alpha;\beta \leq \gamma;\delta \leftarrow \alpha \leq \gamma \wedge [\alpha]\beta \leq \delta$$

$$(=)\ a = b \leftrightarrow a \leq b \wedge b \leq a$$

$$(?)\ (?\phi \leq ?\psi) \leftrightarrow (\phi \rightarrow \psi)$$

$$(ODE)\ (x' = \theta \,\&\, \phi \leq x' = \eta \,\&\, \psi) \leftrightarrow [x' = \theta \,\&\, \phi](x' = \eta \wedge \psi)$$

$$(MP)\ \frac{\phi \rightarrow \psi \quad \phi}{\psi}$$

$$(G)\ \frac{\phi}{[\alpha]\phi}$$

$$(\rightarrow 2\leftrightarrow)\ \frac{\phi \leftrightarrow \psi}{\phi \rightarrow \psi}$$

$$([;])\ [a;\beta]\phi \leftrightarrow [\alpha][\beta]\phi$$

$$(\leq_r)\ \alpha \leq \alpha$$

Fig. 1. Some dRL axioms and rules

KeYmaera X. The theorem prover KeYmaera X [11] implements dL, and more recently its microkernel now also supports its extension dRL [23]. It uses a uniform substitution calculus version [20,23] of the one presented here. Axioms are expressed directly in the logic, with a unique rule – the uniform substitution rule – recovering all its sound instantiations. Uniform substitution is the key to KeYmaera X's small soundness-critical microkernel. Basing the refactoring techniques on the dRL proofs thus provides more automation, while limiting the changes done to the kernel to the new axioms required: Lemma 1 and Lemma 3.

3 Three Levels of Refactoring

The objective of a CPS refactoring is to change a part of a program without affecting the safety of the program or its resulting proof. The reason such change is valid can be justified on three different levels, from a local argument that can be proved automatically to a more general reasoning that requires fixing the proof to accommodate the change. The proofs from this section are accessible in the extended version [25].

3.1 Global Refinement

A refinement between two systems always ensures that a proof using one correctly transfers to a proof of the other. It is highlighted by the axioms $[\leq]$ and $\leq_t$. In both cases, a proof for a refinement $\alpha \leq \beta$ ensures that the formula in blue can be proved by proving the same formula but with the program β instead. Thus, any refactoring from β to α immediately preserves all the corresponding properties, whether about safety or refinement. Note that the axiom $\leq_t$ is equivalent – up-to renaming and propositional reasoning – to $\beta \leq \alpha \rightarrow (\gamma \leq \beta \rightarrow \gamma \leq \alpha)$ which allows reasoning on the right-hand side too, but requires a proof of the converse

refinement: $\beta \leq \alpha$. dRL axiomatization contains all axioms – and thus all facts – about Kleene Algebra with tests, including for instance:

$$(\cup_l)\ \alpha \leq \alpha \cup \beta \qquad \text{(dist-l)}\ (a;b) \cup (a;c) = a;(b \cup c)$$

$$(;_{\text{id-l}})\ ?true;\alpha = \alpha \qquad \text{(dist-r)}\ (a;c) \cup (b;c) = (a \cup b);c$$

Congruence. On their own, the axioms $\leq_t$ and $[\leq]$ are limited to top-level refinements about the whole program. This can be corrected to extend the applicability of global refinement. If the refinement is not on the full program but only a subcomponent, this principle can be extended via general congruence rules. Figure 2 presents one such refactoring extending $[\leq]$ for subcomponents.

$$[\leq]\ \frac{\Gamma \vdash [C(\beta)]\phi \qquad \dfrac{\dfrac{*}{\vdash \alpha \leq \beta}\,\text{Ax}}{\vdots\ \text{ind. on } C}}{\Gamma \vdash C(\alpha) \leq C(\beta)}}{\Gamma \vdash [C(\alpha)]\phi}$$

Fig. 2. Refactor template for box in context via global refinement

Its soundness revolves around the notion of monotone (resp. antitone) contexts. Contexts are expressions (formulas or programs) with a hole. The hole $\cdot_p$ (resp. $\cdot_f$) expects a program (resp. a formula), and we denote by $C(e)$ the expression obtained by substituting the hole by the expression e assuming the kind are respected. Monotone (resp. antitone) contexts are specific contexts that preserve (resp. commute) implications and refinements when applied. The polarity of contexts expresses whether they are monotone ($+$) or antitone ($-$).

Definition 4 (Monotone context). *We define simultaneously monotone and antitone contexts where both the hole and the resulting expression can be formulas or programs as $C^{\pm}_{k_1 k_2}$, where $\pm \in \{+,-\}$ denotes the polarity, $k_1 \in \{f,p\}$ the kind of the resulting expression, and $k_2 \in \{f,p\}$ the kind of the hole.*

$$\begin{aligned}
C^{+}_{kk} &::= \cdot_k \\
C^{\pm}_{pk} &::= \alpha \mid ?C^{\pm}_{fk} \mid x' = \theta \,\&\, C^{\pm}_{fk} \mid C^{\pm}_{pk};C^{\pm}_{pk} \mid C^{\pm}_{pk} \cup C^{\pm}_{pk} \mid C^{\pm}_{pk}{}^{*} \\
C^{\pm}_{fk} &::= \phi \mid C^{\pm}_{fk} \wedge C^{\pm}_{fk} \mid \neg C^{\mp}_{fk} \mid \forall x\, C^{\pm}_{fk} \mid [C^{\mp}_{pk}]C^{\pm}_{fk} \mid C^{\mp}_{pk} \leq C^{\pm}_{pk}
\end{aligned}$$

The kind $k \in \{f,p\}$ specifies the kind of the hole (formula or program), e.g., the hole $\cdot_p$ is a program to program context, i.e., C^{+}_{pp}. $\mp$ stands for the polarity opposite to $\pm$, e.g., for antitone program to formula contexts the grammatical rule for box modalities is $C^{-}_{fp} ::= [C^{+}_{pp}]C^{-}_{fp}$. The only constructs switching polarities are the negation, the box modality (for the program), and the refinement

(for the left-hand side program). It is implicitly extended to derived constructs, e.g., $C^{\mp}_{fk} \to C^{\pm}_{fk}$ as $\phi \to \psi \equiv \neg(\phi \wedge \neg\psi)$. Note that these contexts may not have a hole under an equivalence, e.g., $\cdot_f \leftrightarrow \psi$ would be $(\cdot_f \to \psi) \wedge (\psi \to \cdot_f)$ which is not allowed by the grammar, as both implications have conflicting polarities.

Theorem 1 (Monotone congruence). *Given contexts C^{+}_{pp}, C^{-}_{fp}, C^{-}_{pf} (and similarly for $C^{-}_{pp}, C^{+}_{fp}, C^{+}_{pf}, C^{\pm}_{ff}$), then the following rules are derivable in* dRL*:*

$$\frac{\alpha \leq \beta}{C^{+}_{pp}(\alpha) \leq C^{+}_{pp}(\beta)} \qquad \frac{\beta \leq \alpha}{C^{-}_{fp}(\alpha) \to C^{-}_{fp}(\beta)} \qquad \frac{\psi \to \phi}{C^{-}_{pf}(\phi) \leq C^{-}_{pf}(\psi)}$$

The theorem is proved by induction on the contexts, using axioms like ;≤ to reduce each construct to its subcomponents. The template of Fig. 2 is now a simple application of Theorem 1 for C^{+}_{pp} contexts. Note that even for C_{ff} contexts, the proof may require the use of refinement, for instance with $[?\cdot_f]\phi$ since the hole is in program (itself in a formula). Given the mutually inductive nature of formulas and programs in dRL, having a program counterpart of implications leverages this kind of induction for obtaining much more general principles.

Dealing with Equivalence. We denote by $C_{k_1k_2}$ with $k_1, k_2 \in \{f, p\}$ general contexts, defined by the same grammar as Definition 4 by dropping all polarities. It only ensures the holes are all of the same kind. Since polarities are dropped, $\cdot_f \leftrightarrow \psi$ is a valid C_{ff} context. Compared to refinements, given a global program equivalence, i.e., a proof $\vdash \alpha = \beta$, it is always sound to replace the program α by β in a context, without considering monotonicity or free and bound variables. Program equivalences are dealt with via a specific rule CPE, similar to the dL congruence rules for terms and formulas [19].

Lemma 1 (Contextual program equivalence). *Rule CPE is admissible.*

$$\text{(CPE)}\ \frac{\alpha = \beta}{C_{fp}(\alpha) \leftrightarrow C_{fp}(\beta)}$$

Rule CPE circumvents the need of an induction on C_{fp}, and so the corresponding template for program equivalence, given in Fig. 3, is simpler than for refinement.

$$\text{MP}\ \frac{\Gamma \vdash [C(\beta)]\phi \qquad \to 2\leftrightarrow\ \dfrac{\text{CPE}\ \dfrac{\text{Ax}\ \dfrac{*}{\vdash \beta = \alpha}}{\Gamma \vdash [C(\beta)]\phi \leftrightarrow [C(\alpha)]\phi}}{\Gamma \vdash [C(\beta)]\phi \to [C(\alpha)]\phi}}{\Gamma \vdash [C(\alpha)]\phi}$$

Fig. 3. Refactor template via global program equivalence

Having both these templates transforms simple dRL axioms into refactoring techniques, allowing to freely rewrite hybrid programs during the proofs. For

instance, with dist-l, dist-r, ODE, and $\cup_l$, we immediately obtain formal generic versions of the refactorings (R1–2), (R6–7), and (R9–10) respectively [17], which are given in Fig. 4. Since (R6–7) and (R9–10) are obtained from refinements and not program equivalences, they are only allowed in one direction. Box modality results transfer via (R6) and (R10), while diamond modality results transfer via (R7) and (R9).

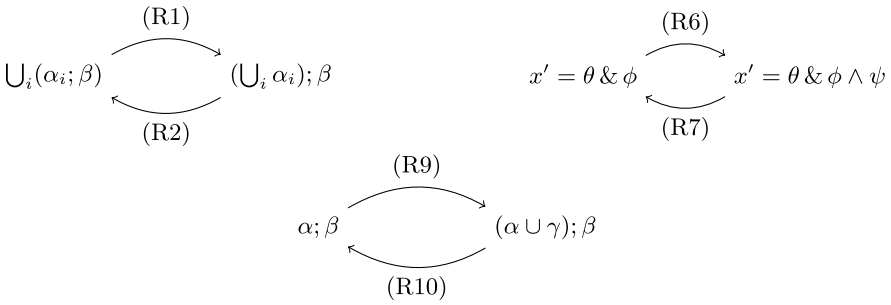


Fig. 4. Program refactorings induced by global refinements

3.2 Local Refinement

Sometimes, local changes are only justified due to the surrounding context. For instance, a test may evaluate to false due to a previous assignment, making one side of a choice $\alpha \cup \gamma$ unreachable. In that case, the other direction for the refactorings (R9–10) of Fig. 4 is correct. To allow these refactorings to occur, we must be able to reason about the local information that is known at the position where the programs lie. This is an issue for the template of Fig. 2 since all context is lost in the congruence, and is answered by considering *local refinements*.

Whereas global refinements enable direct usage of the axioms of dRL, local refinements are useful for applying more complex lemmas, where the refinement or implication holds thanks to some side condition. Compared to the examples of Sect. 3.1, the refinement statements are of the form $\psi \to \alpha \leq \beta$ or $[\gamma]\alpha \leq \beta$. This makes it a more powerful but also more manual refactoring. The conditional refinement may need to be proved by the user beforehand, and the assumption ψ has to hold locally, which must also be proved. Nonetheless, using the proved lemma for refactoring can still be done automatically, similarly to the congruence implementation for Sect. 3.1. The proof is altered to keep the original assumptions Γ, so as to prove that the refinement holds at the position where it is used. From here and onward, we will focus on single-hole program contexts (C^+_{pp}) which will be noted C when unambiguous.

Towards Context-Aware Congruence. To preserve the context, the idea is to look for a congruence property that holds in the same state. Implication-based

and equivalence-based axioms are better suited for this task than proof rules as the conclusion must hold in the same state as the assumption. This makes it possible to use them even if the refinement does not hold globally. In that regards, consider the right subcomponents in $;_{\leq}$. The refinement we obtain ($\beta \leq \delta$) is now under a modality to express the fact that it *only* needs to hold in the states reachable from the current state after any possible execution of the context program α. Given a single-hole context C, we can construct the program that must be executed before the hole using *projective contexts*.

Definition 5 (Projective context). *Given a monotone program context C and a program α, we define the projective context C_α as a formula-to-formula context inductively on the structure of C:*

- *when $C \equiv \cdot_p$, the context C_α is $\cdot_f$*
- *when $C \equiv \gamma; C'$, the context C_α is $[\gamma]C'_\alpha$*
- *when $C \equiv C'^*$, the context C_α is $[C(\alpha)]C'_\alpha$*
- *when $C \equiv C'; \gamma$, or $C' \cup \gamma$, or $\gamma \cup C'$, the context C_α is C'_α*

The dependency on α only shows in loops. Lemma 2 would still hold when using C_β, though it would be less practical in general. Indeed, even though both rules are derivable, the premise with C_β may not hold even when ours is.

Lemma 2. *For a monotone program context C, the following rule is derivable in* dRL*:*

$$\frac{\Gamma \vdash C_\alpha(\alpha \leq \beta)}{\Gamma \vdash C(\alpha) \leq C(\beta)}$$

Using Lemma 2, we obtain a refactoring whose structure is similar to Fig. 2 and shown in Fig. 5, except that the right branch now keeps the assumptions Γ by using the projective context C_α. Closing the goal $\Gamma \vdash C_\alpha(\alpha \leq \beta)$ then depends on the form of the refinement statement used. For statements like $\psi \to \alpha \leq \beta$, it reduces to $\Gamma \vdash C_\alpha(\psi)$ by monotonicity, which still need to be proved. For statements like $[\gamma]\alpha \leq \beta$, it can be concluded immediately when the last modality of C_α is $[\gamma]$, by using G to remove the outer modalities.

$$\lfloor \leq \rfloor \frac{\Gamma \vdash [C(\beta)]\phi \qquad \dfrac{\Gamma \vdash C_\alpha(\alpha \leq \beta)}{\vdots \text{ ind. on } C \text{ (Lemma 2)}} \; \Gamma \vdash C(\alpha) \leq C(\beta)}{\Gamma \vdash [C(\alpha)]\phi}$$

Fig. 5. Refactor template via local refinement

Program Equivalence. In order to preserve local information, program equivalences need to be treated in the same way as local refinements. As the projective context C_α is dependent on the structure of C, the congruence cannot be implemented in just one rule like with Sect. 3.1.

Corollary 1. *For a program context C, the following rule is derivable in* dRL*:*

$$\frac{\Gamma \vdash C_\alpha(\alpha = \beta)}{\Gamma \vdash C(\alpha) = C(\beta)}$$

Refactorings via local refinements are essential as it can be relatively frequent that equivalences that arise from cyber-physical systems rely on generic assumptions like non-negativity of some parameters. Even if generally assumed, the presence of these small assumptions makes the argument inherently local. Section 4.2 shows how that would apply in a realistic setting.

3.3 Ghost Equivalence

The last type of refinements involve introducing new variables. Inside modalities, it is simple to add fresh variables: for instance by [:=], $[x := \theta]\phi \leftrightarrow \phi$ holds whenever x is fresh, and introduces the fresh variable x as a ghost variable. Comparatively, as refinements look at the output value of all variables, its meaning is strongly affected by the addition of new variables. However, it is still possible in dRL to express refinements that disregard the value of a variable: the refinement $\alpha \leq \beta; x := *$ holds when the program α refines β without considering the final value of x. For program equivalences, the corresponding version is symmetric: $(\alpha; x := *) = (\beta; x := *)$, which we will focus on for this section. This technique enables additional refactoring principles:

(ODE_{cst}) $y' = \theta \,\&\, \psi; x := * \;=\; (x := \theta; y' = x \,\&\, \psi; x := *) \quad (x, y \notin \theta; x \notin \psi)$

$(=_{dG})$ $y' = \theta \,\&\, \psi; x := *; x' := * \;=\; (x := c; y' = \theta, x' = \eta \,\&\, \psi; x := *; x' := *)$
(η linear in x, $x \notin \theta, \psi, y$)

The latter program equivalence is the dRL counterpart of dL axiom dG [22].

Lemma 3. *The axioms ODE_{cst} and $=_{dG}$ are sound.*

These new axioms are focused on introducing ghost variables near differential equations. Similar equivalences for assignments can also be used, as long as they are provable in dRL, e.g., $x := \theta; x := * = x := *$.

Congruence. The drawback of the version using refinement is the presence of these nondeterministic assignments. To perform refactorings on a subprogram, these assignments need to be moved in position to ensure that the full program will not be affected by the introduction of the new variable.

Figure 6 showcases the structure to apply a ghost program equivalence of the form $\alpha; x := * = \beta; x := *$. First, we need to introduce the nondeterministic assignment of x, which only succeeds if x is effectively fresh in ϕ. Then, it is

moved next to α. Though the local reasoning is different from the previous refactoring, the general structure is similar and done by deriving the proof by induction on the structure of C. Compared to the previous inductions, the case for the loop is particularly tricky and discussed in more detail below.

Theorem 2. *If x is fresh in C and x is not free in α, then the following program equivalence is derivable in* dRL*:*

$$C(\alpha); x := * = C(\alpha; x := *); x := *$$

Note that it would be unsound to remove the final nondeterministic assignment in general. Indeed, $(c := c + 1; x := *)^*; x := *$ and $(c := c + 1; x := *)^*$ are not equivalent since only the former program can modify x but not c. The program equivalence rule CPE enables the replacement of α by β using the congruence for global refinement from Sect. 3.1. The rest of the refactoring of Fig. 6 uses Theorem 2 again to removing the leftover assignment.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash [C(\beta)]\phi}{\Gamma \vdash [C(\beta); x := *]\phi}\,[:=]}{\dfrac{\vdots\ \text{(Theorem 2)}}{\Gamma \vdash [C(\beta; x := *); x := *]\phi}} \quad \dfrac{\dfrac{*}{\vdash \alpha; x := * = \beta; x := *}\,\text{Ax}}{\ldots}\,\text{CPE}}{\Gamma \vdash [C(\alpha; x := *); x := *]\phi}\,[\leq] \quad \dfrac{\dfrac{*}{\vdots\ \text{ind. on } C \text{ (Theorem 2)}}}{\Gamma \vdash C(\alpha); x := * = C(\alpha; x := *); x := *}}{\Gamma \vdash C(\alpha); x := * \leq C(\alpha; x := *); x := *}\,{=}$$
$$\dfrac{\Gamma \vdash C(\alpha); x := * \leq C(\alpha; x := *); x := *}{\Gamma \vdash [C(\alpha); x := *]\phi}\,[\leq]$$
$$\dfrac{\Gamma \vdash [C(\alpha); x := *]\phi}{\Gamma \vdash [C(\alpha)]\phi}\,[:=]$$

Fig. 6. Refactor template via ghost refinement

Handling Loops. Looking at the induction case for loops, we want to automatically derive $(C(\alpha); x := *)^*; x := * = C(\alpha)^*; x := *$. Due to the repetition, even though the assignment is after the program, it may influence the execution before the next iteration. Thus, the proof of the equivalence above relies on $(x := *; C(\alpha); x := *) = C(\alpha); x := *$, which only holds if x is not free [19, Def.15] in $C(\alpha)$, and is proved by induction on C. Using the template of Fig. 6 with ODE_{cst} or $=_{dG}$ requires applying Theorem 2 on a program that mentions the introduced variable x. To remain applicable, Theorem 2 cannot assume that x is fresh for α like it is for C. This weaker condition holds for both refinements considered, ODE_{cst} and $=_{dG}$, as the right program always start with an assignment on x. This unexpected other induction strengthens the need for having such reasoning automated, as the proof would grow very fast.

4 Applications

This section demonstrates how these refactorings via refinements are applied at various levels of abstraction, from the very concrete implementation in KeYmaera X, to a case study showcasing the proof simplification provided by proved

refactorings, to a more abstract method for proving time-triggered models out of event-triggered models.[1]

4.1 Implementation in KeYmaera X

The source code containing the extension of KeYmaera X for dRL is available online [24]. The implementation consists of two parts. First, the new axioms and rules need to be added to the soundness-critical kernel. This includes Lemma 1, Lemma 3. Additionally, to prevent unsound renaming, the kernel disallows instantiating by a differential variable axioms that are phrased for plain variables (such as in [:=]). This showed a completeness issue in the previous implementation of dRL. As the axiom $=_{dG}$ deals with both kinds of variables, manipulating these nondeterministic assignments also required to duplicate some preexisting axioms with differential variants.

Secondly, the refactorings are implemented as tactics, i.e., automated proofs that combine `provables`, KeYmaera X's certificate of sound inference rules. The tactics are then made accessible to the user via the tactic language [10]. This provides most of the automation to build complex proofs without being soundness-critical. The global refinements from Sect. 3.1 led to the extension of the `useAt` tactic which handles congruence for implications and formula equivalences. Without refinement, congruence proofs for implication were limited to $C^{\pm}_{ff}$ contexts where the hole is not under a program, e.g., $[?\cdot_f]\phi$ was not supported. The tactic now supports all monotone and antitone contexts from Definition 4, by deriving the appropriate sound rule from Theorem 1. Refactorings like Fig. 2 are executed simply by providing the `useAt` tactic with the refinement axiom used as well as the position of α in the sequent. Local refinements are given a specific tactic `focus` which *automatically* computes the projective context and then derives the sound rule of Lemma 2, focusing a refinement down to its core change. The extension to Corollary 1 is similar, but since it is not required for the applications below, it is left as future work. Finally, the proof of Theorem 2 is automatically constructed and interfaced by two tactics, `moveRandomIn` and `moveRandomOut`, that uses Theorem 2 to rewrite the goal in one way or another. Thus, Fig. 6 is done by using [:=] to introduce the fresh variable, `moveRandomIn` to copy it next to α, then applying the ghost equivalence (via `useAt`), before removing the nondeterministic assignment by `moveRandomOut` and [:=].

4.2 Next-Generation Airborne Collision Avoidance System ACAS X

ACAS X is a Airborne Collision Avoidance System developed by the Federal Aviation Administration (FAA). It gives vertical advisories to a pilot to prevent the plane from colliding with other aircraft. Previous formal verification of ACAS X [13] provides hybrid models safe under various assumptions (e.g., pilot's reaction time). Crucially, all these models rely on the definition of the region where an

[1] https://github.com/EnguerrandPrebet/Refactoring-as-Propositions.

advisory is safe. We focus on the first model where a formula L^{-1} expresses that a given advisory will always be safe, i.e., no further advisory will be required. Two formulas expressing that property where given:

- An implicit domain formulation L^{-1}_{impl}, that quantifies over multiple variables including time to ensure that the computed positions of the aircraft are sufficiently far apart. This formula's intent is clear which makes the proof simpler, but the presence of quantifiers makes the effective checking of such formula inefficient and thus not suited for implementation in a real aircraft.
- An explicit domain formulation L^{-1}_{expl}, where the space is split in different modes, leading to a quantifier-free formula, that can be computed quickly in an actual controller.

The ACAS X safety theorem considered has the form

$$(\mathit{init} \wedge L^{-1}) \rightarrow [((\mathit{advisory}; ?L^{-1} \cup \mathit{skip}); \mathit{acc}; \mathit{motion})^*]\mathit{nocol}$$

At each iteration, if not skipped, a new advisory satisfying L^{-1} is chosen, then the acceleration is updated according to the advisory before the two aircraft move in a continuous *motion*. Assuming a safe start (L^{-1}) and some initial condition, the system always ensures the absence of collision *nocol*. This formula was proved using L^{-1}_{impl} [13, Theorem 1]. These two formulations are then proved equivalent assuming *init*, i.e., $\mathit{init} \rightarrow (L^{-1}_{\text{impl}} \leftrightarrow L^{-1}_{\text{expl}})$ [13, Lemma 1]. Because that equivalence is only conditional, dL's congruence rule cannot be used to directly prove the same safety property using L^{-1}_{expl}. The safety transfer from one model to the other had to be done manually. This made the proof exceedingly tedious, with over 200 tactics used.

With local refinements, updating the test from $?L^{-1}_{\text{impl}}$ to $?L^{-1}_{\text{expl}}$ is *one* local refinement which requires a proof of

$$(\mathit{init} \wedge L^{-1}_{\text{expl}}) \rightarrow [((\mathit{advisory}; ?L^{-1}_{\text{expl}} \cup \mathit{skip}); \mathit{acc}; \mathit{motion})^*][\mathit{advisory}]\mathit{init}$$

The condition *init* being composed of a simple fact about the advisory and that some constants are nonnegative (thus a trivial loop invariant), proving the safety of the system using the explicit formulation follows easily. Overall, this decreases the size of the proof to 16 tactics, reducing the need for manual guidance by an order of magnitude.

4.3 Refactoring Event-Triggered Control to Time-Triggered Control

We show how the refactoring techniques presented allow transforming an event-triggered system to a time-triggered system. The overall result is stronger than [15], in addition to being streamlined by the refactorings and formally proved using KeYmaera X. The equivalent model *event* (Fig. 7) from [15] already mentions time, which we circumvent by using the ghost refinement (R2) (Fig. 8).

Event-triggered systems use events as detection mechanisms, which intrinsically assume a continuous sensing from the system: the system regains control

as soon as an event occurs. This behavior makes them simple to prove safe. Conversely, time-triggered systems react on a clock, similar to traditional software. They are thus easier to implement in real systems, but also harder to prove.

Refactoring via refinement enables the transfer of the proof from event-triggered systems to time-triggered systems. Thus, it makes possible to start designing systems as event-triggered, and iterates until a proof of safety is found, before then tackling the more challenging time-triggered system and reusing the insights developed in the first phase without having to restart the proof from scratch. Essentially, bridging the gap between the two type of triggers makes implementable systems also easier to prove.

For demonstrating the capabilities of refinements, we will use the example of a car with position x which must preserve some safety distance in front $x \leq \theta$, and may either brake or accelerate. The continuous dynamics are written *plant* and could be simply expressing the link between acceleration and position, i.e., $x' = v, v' = a$. An event-triggered model for the car may sense the position of x exactly and force the controller to take a decision at the limit, i.e., when $x = \theta$. The hybrid program *event* from Fig. 7 corresponds to this event-triggered model. Having a choice between two continuous dynamics, one with $x \leq \theta$ and one with $x \geq \theta$ ensures that going from $x < \theta$ to $x > \theta$ requires at least two loop iterations, the first one stopping at $x = \theta$, while the second depend on the controller decision. On the other hand, a time-triggered system needs an explicit time t and may only stop at the latest every T seconds. Thus, for this model to be still correct, it should react preemptively, so the condition for accelerating should be stronger, and depends on T. This is expressed in the hybrid program *time* from Fig. 7.

$$\begin{aligned} \mathit{event} &:= ((\mathit{brake} \cup ?x < \theta; \mathit{acc}); (\mathit{plant} \,\&\, x \leq \theta \cup \mathit{plant} \,\&\, x \geq \theta))^* \\ \mathit{time} &:= ((\mathit{brake} \cup ?x < \theta_T; \mathit{acc}); t := 0; \mathit{plant}, t' = 1 \,\&\, t \leq T)^* \end{aligned}$$

Fig. 7. Event and time-triggered systems

As a special case, we will consider the situation where both accelerating and braking are constant – with value A, $-B$ respectively – and the car needs to stop before a position m, meaning the postcondition is $x \leq m$. In that case, $\theta := m - \frac{v^2}{2B}$ and $\theta_T := m - \frac{v^2}{2B} - (\frac{A}{B} + 1)(\frac{A}{2} \cdot t^2 + tv)$ lead to safe systems assuming all constants are positive.

A proof $\phi \rightarrow [\mathit{event}]\psi$ for the event model can be transformed into a proof to the time model $\phi \rightarrow [\mathit{time}]\psi$ via the refactorings in Fig. 8. The first refactoring (R1) removes one of the continuous dynamics. This is done by a single global refinement using $\cup_l$. Refactoring (R2) introduces the new time variable t in the differential equation. It corresponds to a ghost refinement using $=_{dG}$. Compared to (R1), this step will only succeed if t and t' are correctly fresh with regards to the system (e.g., *plant* does not use t) and the postcondition ψ. Refactoring (R3) updates the test condition to take into account for the potential delay T.

$$((brake \cup ?x < \theta; acc); (plant \,\&\, x \leq \theta \cup plant \,\&\, x \geq \theta))^*$$

$$\Big\downarrow \text{ global refinement (R1)}$$

$$((brake \cup ?x < \theta; acc); plant \,\&\, x \leq \theta)^*$$

$$\Big\downarrow \text{ ghost refinement (R2)}$$

$$((brake \cup ?x < \theta; acc); t := 0; plant, t' = 1 \,\&\, x \leq \theta)^*$$

$$\Big\downarrow \text{ local refinement (R3)}$$

$$((brake \cup ?x < \theta_T; acc); t := 0; plant, t' = 1 \,\&\, x \leq \theta)^*$$

$$\Big\downarrow \text{ local refinement (R4)}$$

$$((brake \cup ?x < \theta_T; acc); t := 0; plant, t' = 1 \,\&\, t \leq T)^*$$

Fig. 8. Event-triggered to time-triggered system

It requires as input a proof that $\theta_T \leq \theta$ which is unlikely to hold in general. Thus, we fall back to local refinement, and need to prove that this inequality holds locally. For our specific application, this leads to proving $A > 0 \wedge B > 0 \wedge T > 0 \rightarrow \theta_T < \theta$ which is simple enough to be proved automatically using tools for first-order arithmetic. This lemma proved, local refinement justifies the refactoring if $A > 0 \wedge B > 0 \wedge T > 0$ holds locally, i.e., it is an invariant of the system. Since these are all constants, the proof is trivial. All the manual reasoning above for (R3) only needed to be done on a local basis, as the extension to the full system is automatic. Refactoring (R4) is similar but updates the domain constraint. Since the change occurs inside a differential equation, the resulting lemma is more complex. Indeed, we must show that $[plant, t' = 1 \,\&\, t \leq T]x \leq \theta$ holds locally, i.e., after running the updated controller. This is the essence of the transformation from event-triggered to time-triggered, which has been distilled from the general proof via the automatic congruence reasoning.

Due to the complex handling of nondeterministic assignments for ghost refinements, (R2) has more than half the total number of internal steps used, but the automation ensures it only requires 5–10 manual steps similar to the others.

5 Conclusion

We showed how refinements lead to different notions of formally proved refactoring for hybrid programs. The flexibility of dRL's refactoring-as-propositions principle enables proofs to go beyond global reasoning and to have access to local assumptions when proving refinement of subprograms, without drastically extending its axiomatization. By adding final nondeterministic assignments, we have shown how refinements can suppress the value of some variables, opening the path to ghost refinement as a formal base to allow introduction of fresh variables soundly. These refactoring techniques via refinements are implemented

in the theorem prover KeYmaera X. Since the refactorings presented here rely on refinements for their soundness, they cannot be used directly to prove refactorings that introduce more behaviors to a safe system. Integrating these refactorings formally would require to use the current techniques to isolate the new behaviors—without affecting its overall execution—so that safety is proved by simply looking at the isolated change rather than the full system. Future works include supporting refinements of more complex programs of dL extensions such as communications [6], which are also used in cyber-physical systems.

Acknowledgments. This work has been supported by an Alexander von Humboldt Professorship, the pilot program Core Informatics (KiKIT) of the Helmholtz Association (HGF), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139195881
2. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010). https://doi.org/10.1007/S10009-010-0145-Y
3. Abrial, J.-R., Su, W., Zhu, H.: Formalizing hybrid systems with Event-B. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_13
4. Afendi, M., Laleau, R., Mammar, A.: Modelling hybrid programs with Event-B. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 139–154. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_10
5. Banach, R., Zhu, H., Su, W., Wu, X.: Continuous behaviour in Event-B: a sketch. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 349–352. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_29
6. Brieger, M., Mitsch, S., Platzer, A.: Uniform substitution for dynamic logic with communicating hybrid programs. In: Pientka, B., Tinelli, C. (eds.) CADE 2023. LNCS, vol. 14132, pp. 96–115. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_6
7. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 67–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_5
8. Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K.: Proof-based approach to hybrid systems development: dynamic logic and Event-B. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 155–170. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_11
9. Dupont, G., Ameur, Y.A., Pantel, M., Singh, N.K.: Handling refinement of continuous behaviors: a proof based approach with Event-B. In: 2019 International Symposium on Theoretical Aspects of Software Engineering, TASE 2019, Guilin,

China, 29–31 July 2019, pp. 9–16 (2019). https://doi.org/10.1109/TASE.2019.00-25
10. Fulton, N., Mitsch, S., Bohrer, B., Platzer, A.: Bellerophon: tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 207–224. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_14
11. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
12. Grohe, M., Koskinen, E., Shankar, N. (eds.): Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. ACM, New York (2016)
13. Jeannin, J.-B., et al.: A formally verified hybrid system for the next-generation airborne collision avoidance system. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 21–36. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_2
14. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997). https://doi.org/10.1145/256167.256195
15. Loos, S.M., Platzer, A.: Differential refinement logic. In: Grohe, et al. [12], pp. 505–514. https://doi.org/10.1145/2933575.2934555
16. Lunel, S., Boyer, B., Talpin, J.: Compositional proofs in differential dynamic logic dL. In: 17th International Conference on Application of Concurrency to System Design, ACSD 2017, Zaragoza, Spain, 25–30 June 2017, pp. 19–28 (2017). https://doi.org/10.1109/ACSD.2017.16
17. Mitsch, S., Quesel, J.-D., Platzer, A.: Refactoring, refinement, and reasoning. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 481–496. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_33
18. Platzer, A.: Logics of dynamical systems. In: LICS, pp. 13–24. IEEE, Los Alamitos (2012). https://doi.org/10.1109/LICS.2012.13
19. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reason. **59**(2), 219–265 (2016). https://doi.org/10.1007/s10817-016-9385-1
20. Platzer, A.: Uniform substitution at one fell swoop. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 425–441. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_25
21. Platzer, A.: Hybrid dynamical systems logic and its refinements. Sci. Comput. Program. **239**, 103179 (2025). https://doi.org/10.1016/J.SCICO.2024.103179
22. Platzer, A., Tan, Y.K.: Differential equation axiomatization: the impressive power of differential ghosts. In: Dawar, A., Grädel, E. (eds.) LICS, pp. 819–828. ACM, New York (2018). https://doi.org/10.1145/3209108.3209147
23. Prebet, E., Platzer, A.: Uniform substitution for differential refinement logic. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) IJCAR. LNCS, vol. 14740, pp. 196–215. Springer (2024). https://doi.org/10.1007/978-3-031-63501-4_11
24. Prebet, E., Platzer, A.: Refactoring-as-Propositions: 1.0 (2026). https://doi.org/10.5281/zenodo.20142337
25. Prebet, E., Platzer, A.: Refactoring-as-Propositions: proved refactoring of hybrid systems via proved refinements. CoRR **abs/2605.15001** (2026). https://doi.org/10.48550/arXiv.2605.15001
26. Su, W., Abrial, J., Zhu, H.: Formalizing hybrid systems with Event-B and the Rodin platform. Sci. Comput. Program. **94**, 164–202 (2014). https://doi.org/10.1016/J.SCICO.2014.04.015

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

