A Retrospective on Developing Hybrid System Provers in the KeYmaera Family*

A Tale of Three Provers

Stefan Mitsch[1[0000−0002−3194−9759]] and André Platzer[1[0000−0001−7238−5710]]

Computer Science Department, Carnegie Mellon University, Pittsburgh, USA
smitsch@cs.cmu.edu,aplatzer@cs.cmu.edu

Abstract. This chapter provides a retrospective on the developments of three theorem provers for hybrid systems. While all three theorem provers implement closely related logics of the family of differential dynamic logic, they pursue fundamentally different styles of theorem prover implementations. Since the three provers KeYmaera, KeYmaeraD, and KeYmaera X share a common core logic, yet no line of code, and differ vastly in prover implementation technology, their logical proximity yet technical distance enables us to draw conclusions about the various advantages and disadvantages of different prover implementation styles for different purposes, which we hope are of generalizable interest.

Keywords: History of formal methods · Theorem provers · Differential dynamic logic · Hybrid systems

1 Introduction

Hybrid systems verification is demanding, because the joint discrete and continuous dynamics of hybrid systems bring about significant challenges that merit nothing less than the best support in formal verification. The two primary approaches for hybrid systems verification are model checking based on set-valued search through their state space [3, 15, 18] and theorem proving based on deductive proofs decomposing the system [45, 51].

While a few prior approaches defined parts of hybrid systems in other provers, KeYmaera [56] was the first dedicated theorem prover for hybrid systems. This chapter takes a retrospective on the development of the KeYmaera family of provers for hybrid systems. This is one of the few occasions where the same logic (with only slight variations) has been implemented as theorem provers in widely different styles, enabling us to draw conclusions about the respective advantages and downsides about the different prover implementation styles. With KeYmaera

* This material is based upon work supported by the Air Force Office of Scientific Research under grant number FA9550-16-1-0288 and FA9550-18-1-0120. Any opinions, finding, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

[56] first released on 2007-11-02, KeYmaeraD [62] first released on 2011-10-29, and KeYmaera X [20] first released on 2015-04-18, this chapter is reflecting on experience with more than 12 years of KeYmaera implementation and usage.

The name KeYmaera is a pun on *Chimera*, a hybrid monster from Classical Greek mythology. As the name suggests, KeYmaera is based on the KeY prover [1], which is a dedicated theorem prover for dynamic logic for Java programs covered in many other chapters of this book. Even if that common code basis with KeY was limited to the original KeYmaera prover, the name stuck with its subsequent clean-slate implementations of KeYmaeraD and KeYmaera X.

The third time's the charm, but a lot can be learned about the relative advantages and disadvantages from the different prover designs that may help make informed tradeoffs in other projects for other purposes. KeYmaera is built as a sequent calculus prover for differential dynamic logic [43] making ample use of the taclet mechanism [5] that KeY offers to concisely write proof rule schemata with side conditions and schema variable matching conditions that are checked by Java code. KeYmaeraD is built as a sequent calculus prover for (quantified [46]) differential dynamic logic [43] by directly implementing its rule schemata in the host language Scala. Finally, KeYmaera X is built as a uniform substitution calculus prover for differential dynamic logic [50] whose axioms and proof rules are concrete formulas or pairs of formulas and need no schemata. In addition to their different rule application mechanisms do all three provers reach fundamentally different decisions for their style of basic proof data structures. The experience with the implications of these different prover implementation styles enables us to draw conclusions that we hope to be of generalizable interest.

2 Differential Dynamic Logic and its Proofs in a Nutshell

Differential dynamic logic (dL) [43, 45, 48, 50, 51, 59] supports specification and verification of hybrid systems written in a programming language. Differential dynamic logic provides a direct syntactic representation of hybrid systems along with logical formulas that express properties of their behavior. Polynomials can be used as terms and, with sufficient care about avoiding divisions by zero [10], also rational functions etc. The syntax of *hybrid programs* (HP) is described by the following grammar where α, β are hybrid programs, x is a variable and $e, f(x)$ are terms, Q is a logical formula:

$$\alpha, \beta \ ::= \ x := e \mid ?Q \mid x' = f(x) \,\&\, Q \mid \alpha \cup \beta \mid \alpha; \beta \mid \alpha^*$$

Assignments $x := e$ and tests $?Q$ (to abort execution and discard the run if Q is not true) are as usual. Differential equations $x' = f(x) \,\&\, Q$ can be followed along a solution of $x' = f(x)$ for any amount of time as long as the evolution domain constraint Q is true at every moment along the solution. Nondeterministic choice $\alpha \cup \beta$ runs either α or β, sequential composition $\alpha; \beta$ first runs α and then β, and nondeterministic repetition α^* runs α any natural number of times.

For example, the dynamics depicted in Fig. 1 is modeled with the differential equation system $x'{=}x^2yz$, $y'{=}z$, where x determines the magnitude of repulsive ($x \leq 0$) or attraction force ($x \geq 0$) to the center, y determines the position in the force field, and z controls how quickly to follow the flow (the system stops when $z = 0$). The hybrid program in (1) repeatedly chooses nondeterministically between $z := 0$ to stop flow, or $z := y$ to amplify y in the differential equation:

$$\big((\underbrace{z := 0}_{\text{stop flow}} \cup \underbrace{z := y}_{\text{amplify}}); \ \underbrace{\{x'{=}x^2yz,\ y'{=}z\}}_{\text{ODE, see Fig. 1}}\big)^* \quad (1)$$

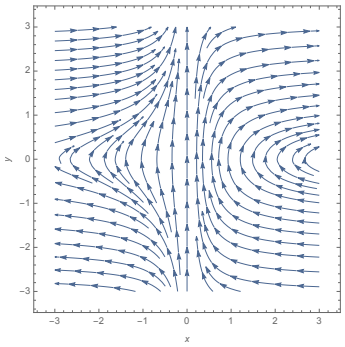


Fig. 1: Flow of differential equation $x'{=}x^2yz$, $y'{=}z$ for $z = 1$

It is important that the right-hand sides of assignments are not simply substituted into the ODE in all cases, since a wrong substitution $z := y$ results in a vastly different ODE $x' = x^2y^2, y' = y$. The assignments $z := 0$ and $z := y$ in (1) pick values for z: z is either 0 or equal to the value of y at the time of the assignment (in this case, equal to the value of y at the beginning of the ODE). Also, the value of z in this example stays constant throughout the ODE, since there is no z' and so implicitly $z' = 0$.

The formulas of dL describe properties of hybrid programs. *Formulas of differential dynamic logic* are described by the following grammar where P, Q are formulas, $e, \tilde{e}$ are terms, x is a variable and α is a hybrid program:

$$P, Q \ ::= \ e \geq \tilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x\, P \mid \exists x\, P \mid [\alpha]P \mid \langle\alpha\rangle P$$

The operators of first-order real arithmetic are as usual with quantifiers ranging over the reals. For any HP α and dL formula P is $[\alpha]P$ true in a state iff P is true after all ways of running α. Dually, $\langle\alpha\rangle P$ is true in a state iff P is true after at least one way of running α. The former is particularly useful for stating safety properties while the latter is useful for stating liveness properties, but their mixed use is powerful as well. The semantics and axiomatization of dL are described elsewhere [43, 45, 47, 48, 50, 51, 59]. For this chapter, it suffices to understand dL informally and remember, e.g., that dL formula $P \rightarrow [\alpha]Q$ is valid (true in all states) iff the postcondition formula Q holds after all runs of hybrid program α that start in initial states satisfying assumption formula P.

For example, the conjecture that $x \geq 0$ always remains true after running the program (1) if $x \geq 0$ was true before is expressed in dL formula (2):

$$\underbrace{x \geq 0}_{\text{assumption formula } P} \rightarrow \overbrace{[\big((z := 0 \cup z := y);\ \{x'{=}x^2yz,\ y'{=}z\}\big)^*]}^{\text{all runs of hybrid program } \alpha} \underbrace{x \geq 0}_{\text{postcondition formula } Q} \quad (2)$$

The dL sequent calculus [43,45,51] works with *sequents* of the form $\Gamma \vdash \Delta$ for a finite set of formulas Γ (*antecedent*) and a finite set of formulas Δ (*succedent*).

$$([\cup])\ [\alpha \cup \beta]P \leftrightarrow [\alpha]P \wedge [\beta]P$$

$$([\cup]\mathrm{R})\ \frac{\Gamma \vdash [\alpha]\phi, \Delta \quad \Gamma \vdash [\beta]\phi, \Delta}{\Gamma \vdash [\alpha \cup \beta]\phi, \Delta}$$

$$([\cup]\mathrm{L})\ \frac{\Gamma, [\alpha]\phi, [\beta]\phi \vdash \Delta}{\Gamma, [\alpha \cup \beta]\phi \vdash \Delta}$$

$$([:=])\ [x := e]p(x) \leftrightarrow p(e)$$

$$([;])\ [\alpha;\beta]P \leftrightarrow [\alpha][\beta]P$$

$$(\text{loop})\ \frac{\Gamma \vdash J, \Delta \quad J \vdash [\alpha]J \quad J \vdash P}{\Gamma \vdash [\alpha^*]P, \Delta}$$

$$(\text{dI})\ \frac{\Gamma \vdash F, \Delta \quad Q \vdash [x' := f(x)](F)'}{\Gamma \vdash [x' = f(x)\,\&\,Q]F, \Delta}$$

Fig. 2: Some dL axioms and proof rules

The meaning of sequent $\Gamma \vdash \Delta$ is that of dL formula $(\bigwedge_{P \in \Gamma} P) \rightarrow (\bigvee_{Q \in \Delta} Q)$. All assumptions are listed in Γ, the set of alternatives to prove are in Δ.

Some typical axioms and proof rules of dL are listed in Fig. 2. The axiom $[\cup]$, for example, expresses that any formula of the form $[\alpha \cup \beta]P$ is equivalent to the corresponding conjunction $[\alpha]P \wedge [\beta]P$. The proof rules $[\cup]$R and $[\cup]$L provide corresponding decompositions of $[\alpha \cup \beta]P$ into $[\alpha]P$ as well as $[\beta]P$ when $[\alpha \cup \beta]P$ is a succedent formula on the right or an antecedent formula on the left in sequent calculus, respectively. The $[\cup]$R rule is easiest to understand without Δ, but whenever Δ is true, the conclusion is true whether or not $[\alpha \cup \beta]P$ is. The loop proof rule expresses that for proving the conclusion $\Gamma \vdash [\alpha^*]P, \Delta$ below the rule bar it suffices to pick an invariant formula J and show that J is initially true or Δ (left premise above rule bar), that J remains true after any run of the loop body α (middle premise), and that J implies the desired postcondition P (right premise). The differential invariant proof rule dI expresses that F is always true after a differential equation $x' = f(x)\,\&\,Q$ if it is true initially (left premise) and its differential $(F)'$ is true when Q is after assigning the right-hand side $f(x)$ of the differential equation to its left-hand side x' (right premise). The differential formula $(F)'$ intuitively expresses that F locally remains true when following the differential equation $x' = f(x)$. Rule dI makes it possible to prove properties of ODEs without having to solve them [59].

Figure 3 illustrates how these axioms and proof rules are used in a sequent proof of the dL formula (2). In a sequent proof, the conclusion (below the horizontal bar) follows from the premises (above the bar), justified by the axiom or proof rule annotated to the left of the bar. The first step at the bottom of Fig. 3 ($\rightarrow$R) makes the left-hand side of the implication available as assumption. Next, the loop rule splits the proof into three subgoals: the left premise (base case) and the right premise (use case) close, because their succedent holds by antecedent assumption, the middle premise (induction step) details are listed in Fig. 3a.

In the proof of the middle premise (induction step), after splitting the sequential composition by axiom $[;]$, the nondeterministic choice axiom ($[\cup]$) splits the proof into two separate modalities. The $[:=]$ axiom applied to the assignment $z := 0$ simply replaces the occurrences of z with 0. In the second assignment $(z := y)$, however, the right-hand side cannot simply replace the left-hand side, since it is bound in the subsequent differential equation; therefore, we split off

$$
\dfrac{\dfrac{\dfrac{}{x \geq 0 \vdash x \geq 0}\,*}{x \geq 0 \vdash [\{x'=x^2y\cdot 0,\ y'=0\}]x \geq 0}\,\text{dI} \qquad \dfrac{\dfrac{}{\vdash 0 \geq 0}\,*}{\vdash [x' := x^2y\cdot 0][y' := 0]x' \geq 0}\,[:=] \qquad \ldots}{\dfrac{\dfrac{\dfrac{\dfrac{x \geq 0 \vdash \begin{array}{l}[\{x'=x^2y\cdot 0,\ y'=0\}]x \geq 0\\ \quad \wedge [z := y][\{x'=x^2yz,\ y'=z\}]x \geq 0\end{array}}{x \geq 0 \vdash \begin{array}{l}[z := 0][\{x'=x^2yz,\ y'=z\}]x \geq 0\\ \quad \wedge [z := y][\{x'=x^2yz,\ y'=z\}]x \geq 0\end{array}}\,[:=]}{x \geq 0 \vdash [(z := 0 \cup z := y)][\{x'=x^2yz,\ y'=z\}]x \geq 0}\,[\cup]}{x \geq 0 \vdash [(z := 0 \cup z := y);\ \{x'=x^2yz,\ y'=z\}]x \geq 0}\,[;]}\,\wedge\text{R}
$$

(a) Proof of middle premise of rule loop (induction step)

$$
\dfrac{\dfrac{\dfrac{}{x \geq 0 \vdash x \geq 0}\,* \qquad \text{Induction step (3a)} \qquad \dfrac{}{x \geq 0 \vdash x \geq 0}\,*}{x \geq 0 \vdash [((z := 0 \cup z := y);\ \{x'=x^2yz,\ y'=z\})^*]x \geq 0}\,\text{loop}}{\vdash x \geq 0 \to [((z := 0 \cup z := y);\ \{x'=x^2yz,\ y'=z\})^*]x \geq 0}\,\to\text{R}
$$

Fig. 3: Sequent proof of dL formula (2)

this branch using ∧R and omit further steps, but will return to the discussion of such assignments in later sections. Now the differential equation shape fits the dI rule: the resulting left branch shows that invariant $x \geq 0$ is true initially, while the right branch shows that the differential $(x \geq 0)'$, which expands to $x' \geq 0$, is true after assigning the right-hand sides of the differential equation.

The provers, KeYmaera 3, KeYmaeraD, and KeYmaera X differ fundamentally in terms of how they implement the dL proof calculus, which also manifests in differences in the concrete mechanics of such sequent proofs, as will be shown below. Only KeYmaera X uses axioms such as [∪] while KeYmaera 3 and KeYmaeraD implement sequent calculus rules similar to [∪]R,[∪]L instead. KeYmaera 3 provides a proof rule schema mechanism in which the rules are implemented while KeYmaeraD implements sequent calculus proof rules directly using the pattern matching variables of the language it is implemented in. A full list of axioms and proof rules of dL is provided elsewhere [43, 45, 50, 51, 59].

3 KeYmaera 3 – A Big Prover With a Big Heart

The KeYmaera 3 theorem prover[1] [56] implements the original sequent calculus of differential dynamic logic for hybrid systems [42, 43, 45] in a mix of Java and some Scala by extending the KeY prover, a prover for Java programs.

Design Principles. The rationale behind the design of KeYmaera is that the quickest way of getting a theorem prover for hybrid systems working is to build it on top of another successful prover. As a dynamic logic prover with a

[1] KeYmaera versions 1–3 are available at http://symbolaris.com/info/KeYmaera.html

well-established prover infrastructure and sophisticated user interface, the KeY prover [1, 2, 6] is the canonical choice. On the highest level of abstraction, the design of KeYmaera needed the programming language Java within dynamic logic modalities of KeY to be replaced by hybrid programs. Of course, semantical changes required a move from the arithmetic over integers and object-oriented data types that are dominant in Java programs to the arithmetic over reals for hybrid systems instead. But the hope was that KeY's rule application mechanism with taclets (lightweight, soundness-critical tactics that can invoke soundness-critical code, see [5]) and propositional/first-order logic calculus automation could be retained, and all the considerable effort that went into the nontrivial user interface design of KeY could be reused directly for hybrid systems.

Implementation Realities. For KeYmaera 3, the dL sequent calculus rule [∪]R for programs starting with a nondeterministic choice would be written as:

$$([\cup]\mathrm{R})\ \frac{\vdash [\alpha]\phi \quad \vdash [\beta]\phi}{\vdash [\alpha \cup \beta]\phi}$$

KeY provides *taclets* [5], which combine proof rules describing what syntactic element is replaced by which other transformed element with grouping into proof rule priorities. The [∪]R rule is implemented in KeYmaera 3 as a taclet:

```
1 box_choice_right {
2   \find (==> \modality{#boxtr}#dl ++ #dl2\endmodality(post))
3   \replacewith (==> \modality{#boxtr}#dl\endmodality(post));
4   \replacewith (==> \modality{#boxtr}#dl2\endmodality(post))
5   \heuristics (simplify_prog)
6   \displayname "[++] choice"
7 };
```

Taclet `box_choice_right` uses `\find` to search for a succedent formula (as indicated by the occurrence in the right-hand side of the sequent turnstyle ⊢ rendered as `==>` in ASCII) with a [·]-type modality whose top-level program is of a nondeterministic choice form `#dl` ∪ `#dl2` (where ∪ is rendered as `++` in ASCII, and `#dl` and `#dl2` are hybrid programs) and any formula `post` as a post-condition, see Line 2. Each such occurrence will be replaced with two premises in which the corresponding active formula is, instead, replaced by the same [·]-type modality and the same postcondition formula that matched `post` in `\find` and with the hybrid program that matched `#dl` as the hybrid program on the first premise (Line 3) and with the hybrid program that matched `#dl2` as the hybrid program on the second premise (Line 4), respectively. The declaration `\heuristics (simplify_prog)` in Line 5 assigns the use of this taclet to the group `simplify_prog` that the proof strategy uses with higher priority than expensive proof rules. The `\displayname` in Line 6 shows what name is used for the rule on the user interface. The presence of the schema variable character `#` highlights that an internal matching algorithm will be run in Java to check that

what occurs in place of `#dl` actually is a hybrid program and what occurs in place of `#boxtr` actually is a box-type modality, etc. There is a second taclet that handles nondeterministic choices in box modalities in the antecedent, and two more taclets that handle nondeterministic choices in diamond modalities.

The proof strategy mechanism that KeYmaera 3 inherits from KeY computes the priorities of the heuristic groups of all taclets after instantiating them to the formulas at hand and checking for their applicability. It favors the use of high priority taclet uses over lower priority taclets. For fair rule selection, lower priority taclets gain priority over time if they have been applicable to the same formula in a goal for a long time. When selecting preexisting heuristic groups such as `simplify_prog`, taclets will immediately be used by the automatic proof strategy with the priority associated to that heuristic group. That makes it easy to add simple proof rules to the proof automation. More advanced proof strategies that conditionally use proof rules or use proof rules in succession are much more difficult to encode with priorities. To understand where succession of proof rules comes in, think of an assumption $\forall x\,(x = e \rightarrow \phi)$ where the quantifier first wants to be instantiated to e and then the resulting equality is subsequently rewritten. Those cases need new strategy features implemented in Java that add or subtract cost for the applicability of a rule as a function of the state of the proof to make it cheaper for proof strategies to apply the taclets in the intended order. For example with a strategy feature that gives equality rewriting a low cost if the proof tree has previously instantiated a universal quantifier of an implicational formula conditioned on an equation for the quantified formula. Disproving formulas by exhaustion of rule applications is not feasible with such an approach because many different orderings of applicable proof rules would have to be considered, so extraneous mechanisms for disproving are required [63].

Successes. The hope of enabling a quick implementation for hybrid systems theorem proving by building it on top of KeY was largely met. KeY's user interface also proved to be extremely valuable to quickly get visual feedback on what proof search was doing or where rule applications went wrong. KeY's taclet mechanism provides a mechanism to quickly capture proof rules and proof strategy hints simultaneously in a single place. This makes it easy to add rules to the prover that are automatically applied by proof search with simple customization of the priority with which a rule group is used. The direct implementation style of the taclet application mechanism also makes it reasonably fast. With moderate effort this leads to a hybrid systems prover that is able to prove moderate-complexity problems with user-provided invariants and solvable differential equations. More sophisticated problems needed more sophisticated invasive changes to the prover to become scalable, but starting from KeY certainly got KeYmaera 3 a leg up.

By design, KeY manages a single proof tree on which a single proof strategy sequentially executes proof rules at a leaf whose premises will be added as children; KeYmaera inherits this proof management. This explicit proof data structure can be displayed directly in the user interface and makes it possible to traverse the proof tree if a proof strategy feature wants to understand the struc-

ture of the proof to determine the priority of a possible taclet application. In fact, traversal of the proof tree is often crucial to strategically coordinate taclet uses with one another, because, besides rule cost and heuristic caches, the proof tree is the only means of communication from one taclet application to another.

After major revisions and additions to the proof strategies with conditional proof rule application and sequencing, as well as means of conducting separate hypothetical proofs and caching, it was possible to achieve significant proof automation in KeYmaera 3, including fixedpoint-based invariant generation [54].

KeYmaera provides a pragmatic combination of built-in proof automation and click-based user interaction. The underlying KeY prover manages a single proof and makes it possible to interrupt proof automation at any intermediate stage. That way, users can complement deficiencies of proof automation by inspecting how far automation got, then backtracking to some intermediate proof step before the automation went off the rails, help by selective manual interaction, and then yield control to proof automation again. Anecdotally, this capability was one of the most admired features beyond the full automation that KeYmaera provided. The downside is that novice users, amid the overwhelmingly large number of automatically created proof obligations, often loose track of the remaining proof effort and how it relates to the original proof goal, and whether it would be better to undo automatic proof steps to limit excessive branching.

Challenges. More challenging than anticipated was the correct rendition of differential dynamic logic sequent calculus rules as taclets. Taclets provide flexibility and apply rules in more general scenarios without having to explicitly state them. This makes it significantly easier to write taclets. But one of the unintended consequences of that taclet application mechanism is precisely that rules can be applied in settings that are more general than what one might have had in mind. What is useful for locally sound proof rules such as $[\cup]$R can easily create soundness problems for complicated rules. The taclet for $[\cup]$R can be used even in scenarios that were not explicitly programmed in, for example under update contexts. Thanks to KeY's taclet application mechanism, the rule $[\cup]$R can automatically be applied in a more general style in any additional sequent context $\Gamma \vdash \Delta$ with any finite set $\mathcal{U}$ of parallel assignments (called *update* [2,7,43]) that later simultaneously change the values of the affected variables:

$$[\cup]\mathrm{R}\frac{\Gamma \vdash \mathcal{U}[\alpha]\phi, \Delta \quad \Gamma \vdash \mathcal{U}[\beta]\phi, \Delta}{\Gamma \vdash \mathcal{U}[\alpha \cup \beta]\phi, \Delta}$$

But the taclet application mechanism can quickly cause unsoundness for complicated rules such as loop invariants when the taclet implementor does not explicitly consider all possible generalizations. A loop invariant proof rule for nondeterministic repetitions in KeYmaera 3 could be phrased, e.g., as:

$$(\mathrm{loopR}_{\not z})\ \frac{\vdash J \quad J \vdash \phi \quad J \vdash [\alpha]J}{\vdash [\alpha^*]\phi}$$

But it would be entirely incorrect if the taclet application mechanism also wraps rule loopR↯ into a sequent or update context:

$$\text{loopR}\lightning\ \frac{\Gamma \vdash \mathcal{U}J, \Delta \quad \Gamma,\mathcal{U}J \vdash \mathcal{U}\phi, \Delta \quad \Gamma,\mathcal{U}J \vdash \mathcal{U}[\alpha]J, \Delta}{\Gamma \vdash \mathcal{U}[\alpha^*]\phi, \Delta}$$

Every single occurrence of Γ or Δ or $\mathcal{U}$ in the second or third premise causes unsoundness, so such generalization needs to be prevented at all cost!

Due to its more delicate nature, the loop invariant rule needs more care when phrased as a taclet to avoid inadvertent taclet generalization like loopR↯ above:

```
1  loop_inv_box_quan {
2    \find (==> \modality{#boxtr}#dl*\endmodality(post))
3    "Invariant Initially Valid":
4      \replacewith (==> inv);
5    "Use Case":
6      \replacewith (==> #UnivCl(\[#dl\]true, inv->post, false));
7    "Body Preserves Invariant":
8      \replacewith (==> #UnivCl(
9          \[#dl\]true,
10         inv -> \modality{#boxtr}#dl\endmodality(inv),
11         true))
12   \heuristics (loop_invariant, loop_invariant_proposal)
13   \onlyRigidFunctions
14   \displayname "ind loop invariant"
15 };
```

This taclet implements the loop invariant rule loop with generalization resilience:

$$(ind') \ \frac{\vdash \phi \quad \vdash \forall^\alpha(\psi \to [\alpha]\psi) \quad \vdash \forall^\alpha(\psi \to \phi)}{\vdash [\alpha^*]\phi}$$

The universal closure operator $\forall^\alpha$ forms the universal closure $\forall x_1 \ldots \forall x_n$ where $x_1, \ldots, x_n$ are all variables bound / written to by the hybrid program α. The `#UnivCl` meta operator[2] in Lines 6 and 8 (abbreviating `#dlUniversalClosure`) is implemented in 788 complex lines of Java and computes the inverse transitive closure of the operators of the variable dependencies of the modality passed in as its first argument over the formula passed in as its second argument and optimizes some of those dependencies depending on the third argument. The two alternative names used in the `\heuristics` group in Line 12 are arbitrary but used as hooks in the priority-based proof strategy implementation to trigger the use of the best rules under the present circumstances. The meta operators that introduce universal closures are crucial to make the implicit taclet generalizations sound when they wrap the rule into a sequent or update context $\Gamma, \Delta, \mathcal{U}$:

$$ind'\ \frac{\Gamma \vdash \mathcal{U}J, \Delta \quad \Gamma \vdash \mathcal{U}\forall^\alpha(J \to \phi), \Delta \quad \Gamma \vdash \mathcal{U}\forall^\alpha(J \to [\alpha]J), \Delta}{\Gamma \vdash \mathcal{U}[\alpha^*]\phi, \Delta}$$

[2] The meta operator introduces $\forall X\,[x := X]$, because KeY and KeYmaera distinguish categories of variables. They do not allow quantification over program variables x and do not allow assignment to logical variables X, so a mix with both is needed.

Any changes in update $\mathcal{U}$ to bound variables of α will be overwritten by the universal quantifiers of the closure $\forall^\alpha$. Likewise any assumptions about the initial state that reside in Γ, Δ will become obsolete by the quantifiers of $\forall^\alpha$. A downside of this approach is the abundance of irrelevant formulas it leaves around in Γ, Δ. This also makes the proof steps harder to trace except for expert users. Finally, the `\onlyRigidFunctions` constraint in Line 13 triggers a search through the matched formulas to check no assignable function symbols occur, which would render the usage of the taclet unsound.

The differential invariants proof rule dI of Fig. 2 is implemented as a taclet:

```
1  diffind {
2    \find (==> \[#normODE\]post)
3    \varcond (\isFirstOrderFormula(post))
4    "Invariant Initially Valid":
5      \replacewith (#InvPart(\[#normODE\]post) ==> post);
6    "ODE Preserves Invariant":
7      \replacewith (==> #UnivCl(
8          \[#normODE\]true,
9          #DiffInd(\[#normODE\]post),
10         false))
11   \heuristics (invariant_diff, diff_rule)
12   \onlyRigidFunctions
13   \displayname "DI differential invariant"
14 };
```

Upon each use of this taclet, the metaoperator `#InvPart` in Line 5 triggers a computation to extract the invariant from the differential equation, whose schema variable `#normODE` has a matching algorithm to check that the concrete differential equation is given in normalized form (so a list of equations with the only derivatives occurring once as an isolated variable on the left-hand side and a single formula without derivatives as evolution domain constraint). The meta operator `#UnivCl` for universal closure that was already used in the loop taclet is important for soundness also in the differential invariants taclet, which additionally uses a meta operator `#DiffInd` in Line 9 for computing the differential invariant condition for the postcondition `post` as a function of the differential equation matched by `#normODE` [44, 45]. The presence of the `\varcond` in Line 3 additionally indicates that a meta operator will be run before using the taclet to check that the postcondition `post` is a first-order logic formula. Almost all of the insights behind differential invariants in KeYmaera are provided in the opaque implementation of the meta operators. The meta operators for the `diffind` taclet are implemented in 2413 lines of Java and Scala code.

Assignments can be handled very easily with taclets in KeYmaera but the main reason is that they are immediately converted to updates `{#dlx:=#dle}` who have their own built-in application algorithm called update simplifier:

```
1  assignment_to_update_right {
2    \find (==> \modality{#allmodal}#dlx:=#dle\endmodality(post))
3    \replacewith (==> {#dlx:=#dle} post)
4    \heuristics(simplify_prog)
```

```
5   \displayname ":= assign"
6 };
```

While the taclets give a general idea of the way how a proof rule is applied, most of their more subtle soundness-critical aspects are hidden in the Java implementation of their corresponding schema matching and meta operators.

Modest modifications of the automatic proof search strategy that derives from assigning taclets to strategy groups are easy by changing the priority with which the strategy group of a taclet is being applied. More involved changes of proof search procedures cannot be encoded well with rule priorities but need major invasive changes to KeYmaera's proof strategies. Fixedpoint-based invariant search procedures [54] required multiple proofs to be formed and quickly discarded if unsuccessful, which is counter to the sequential intention of proof search within a proof tree in KeYmaera. Every time the proof search strategy needs to decide whether to apply, say, an invariant proof rule with a member J_1 of a stream of generated invariants, it would start a new *hypothetical proof* in the background to see if that proof with that invariant candidate J_1 would succeed. And if it does succeed, the prover discards the entire hypothetical proof (hence the name) and commits to making the first proof step with that invariant J_1, otherwise it tries a hypothetical proof with another invariant candidate J_2.

The downside of such a sequential emulation of parallel proof exploration in KeYmaera's sequential proof engine is the large number of repeated steps. This is especially true when nested loops and differential equations occur in more complex hybrid systems, so that an entire nested chain of proofs needs to be finished and then discarded to commit to the first proof step; most of the discarded steps need to be repeated later at the next important turn of the proof. Performance mitigation includes a cache that remembers successful and unsuccessful proof attempts of propositionally similar questions at choice points to better guide proof search in the future or in later nested hypothetical proofs.

KeYmaera implements a complete deduction modulo theories approach using free variables and Skolemization with invertible quantifiers and real quantifier elimination [43]. This approach is instrumental for handling complicated modality/quantifier nestings and automatically enables sound quantifier shifts while preventing unsound quantifier rearrangements. The downside is that this requires a proof-global proof rule for real existential quantifiers over modal formulas.

Both full automation and click-based user interaction are great for novice users exploring medium complexity problems. More difficult problems come with more substantial challenges that are more likely out of reach for fully automatic proof strategies yet too tedious to click through manually.

Another challenge with the design of KeYmaera is the absence of proof tactics or proof scripts and the fact that all attempts of adding them rendered them soundness-critical. It is simple to choose from among KeYmaera's predefined proof strategies but so hard to add custom proof automation that only three people succeeded. Even conceptually simple customizations such as noncanonical decompositions inside out are hard to implement but can have a huge impact on keeping the branching factor down to save millions of proofs steps [60, 61].

Explicit storage of the proof tree with all proof steps makes it easy to inspect and navigate what was done. Likewise exhaustive applicability checks for taclets at every proof step make it convenient for users to determine which proof rules make sense to try. Both decisions are adverse for large proofs with large formulas, which causes nontrivial memory pressure and wastes an enormous amount of time with identifying at every step which rules are applicable where and how and at what cost. These effects accumulate so that KeYmaera 3 may need an hour per proof step in larger proofs. Storing the full proof tree also requires a lot of reproving effort when loading a (full or partial) proof, often taking a few hours on complex case studies. This is an example where different prover design choices are needed depending on the expected size of proofs.

The biggest downside of the KeYmaera prover design is that its general infrastructure and proof mechanisms come at the cost of leading to quite a large soundness-critical prover kernel. While this kernel is isolated from the rest of the prover, its taclet infrastructure, schematic matching mechanisms, and built-in operators not only give the kernel a very central flavor in the entire prover implementation but also lead to its roughly 136k lines of soundness-critical code written in a mix of mostly Java and some Scala as well as more than 2000 taclets.

Outcomes. Major case studies verified in KeYmaera include safety of a roundabout aircraft collision avoidance system [55], safety, controllability, reactivity and liveness of the European Train Control System ETCS [57], adaptive cruise control systems for cars [28], robot obstacle avoidance scenarios [32], basic safety for the Next-generation Airborne Collision Avoidance System ACAS X [23], and an adversarial robotic factory automation scenario [61]. KeYmaera is embedded into the hybrid systems modeling environment Sphinx [34], which provides UML-style graphical and textual modeling of hybrid programs, and interacts with KeYmaera to discharge proof obligations in dL.

These case studies confirm that differential dynamic logic is versatile for verifying even complicated hybrid systems. They reassure that KeYmaera has solid automation capabilities and makes it possible to interactively verify complicated systems out of reach for automation. What they also confirm is the nuisance of repetitive interaction that users face when working on complicated applications far out of reach for the small selection of automatic proof strategies that ship with KeYmaera. Especially model changes require tedious repetitions of interactive proofs. KeYmaera proofs store every proof step in detail, so are not a very practical source when redoing proofs making verification results hard to modify.

Other takeaways include the loss of traceability for non-experts caused by the additional quantifiers, updates, and static-single-assignment variable renaming that KeYmaera introduces to make local taclet applications sound. Inexperienced users easily get lost in the proof tree and fail to exercise appropriate branching control, which significantly increases verification complexity. Even experienced users may get lost in the details of a complete proof tree with all steps while missing contextual knowledge when projecting out intermediate proof steps.

What Else KeYmaera 3 Offers. Beyond the scope for this chapter is the fact that KeYmaera 3 also implements the extensions of differential-algebraic dynamic logic with differential algebraic equations [44], differential temporal dynamic logic with safety throughout [45], differential dynamic game logic that adds separate game constructs on top of `dL` [61], and was finally also extended to implement quantified differential dynamic logic for distributed hybrid systems [46]. KeYmaera links to a large number of real arithmetic decision procedures and even has a built-in implementation for simple real arithmetic [58]. The implementation of differential temporal dynamic logic is particularly parsimonious in KeYmaera 3 as, e.g., the `#boxtr` schema variable used in taclet `box_choice_right` can match on either `dL`'s box modality or the throughout modality of differential temporal dynamic logic. One taclet implements two rules.

4 KeYmaeraD – An Experiment in Distributed Theorem Proving With Direct Control

The KeYmaeraD hybrid systems theorem prover[3] [62] is a bare-bones prover with a direct rendition of the differential dynamic logic sequent calculus [43] (with extensions for distributed hybrid systems [46]) in Scala.

Design Principles. Beyond supporting *distributed* hybrid systems, KeYmaeraD was designed with the primary motivation of overcoming inherent scalability limits caused by the sequential prover design of KeYmaera 3. Rather than having a single proof tree explored sequentially as in KeYmaera 3, KeYmaeraD supports a more general AND/OR proof tree where some nodes are AND-branching (all subgoals need to be proved, it suffices to disprove one) and other nodes are OR-branching (one subgoal needs to be proved, or all need to be disproved). In order to favor execution speed and retain a lightweight implementation, KeYmaeraD does not provide a rule application mechanism such as KeYmaera 3 taclets but entirely relies on the pattern matching capabilities of the host language (Scala). KeYmaeraD supports a built-in computation unit abstraction that is used to decide which part of the *parallel* AND/OR proof tree to explore next. In terms of user interaction, KeYmaeraD provides basic AND/OR proof tree rendering with minimalistic goal printing and interaction with the Scala REPL for rule and tactic application, but does not provide other UI infrastructure like KeYmaera 3 and KeYmaera X to filter rules by applicability or to suggest proof steps.

Successes. KeYmaeraD allows direct programming of proofs in the Scala host language and provides ways of combining them with simple tactic combinators in Scala. Its tactic library captures direct proof rule application but does not implement a sophisticated tactic library combining inferences to achieve higher-level reasoning. The AND/OR proof tree of KeYmaeraD emphasizes parallel

[3] KeYmaeraD is available at http://symbolaris.com/info/KeYmaeraD.html

proof search to explore different proof options, which provides significant opportunities for quickly discovering proofs. For example, KeYmaeraD can efficiently construct proofs for a given list of loop invariants in parallel. This is a very powerful mechanism for medium complexity examples where the computational demand does not significantly exceed the computational resources. The direct implementation in a host language ensures low computational cost per rule application. KeYmaeraD is also the first ever prover for distributed hybrid systems.

Challenges. The biggest downside of KeYmaeraD's prover design is its inflexible rule application, limited to top-level sequent calculus uses. Where KeYmaera 3 is sometimes overly permissive, KeYmaeraD is overly narrow-minded in its proof rules and only supports one style of proofs. For example, it would be impossible to avoid exponential blowups that some proofs face unless working inside out [37,61]. While Scala gives a lot of flexibility, the downside is that proofs can only be conducted by programming them in Scala or typing them into a command line in Scala's read-eval-print-loop (REPL), both of which require expert knowledge about the prover's internal implementation details. The simplistic user interface only renders the proof tree and expects REPL commands to apply proof rules with a `goto` command to select the proof tree node, thereby making it comparably hard for novices to get started. Since everything is implemented explicitly in Scala, it is easy to change the prover, but KeYmaeraD does not provide sufficient soundness protection because users can create arbitrary new proof rules in any arbitrary part of the code or they could annotate incorrect solutions of differential equations that KeYmaeraD will use without checking.

Implementation Realities. KeYmaeraD implements the [∪]∧L,[∪]∧R sequent calculus proof rule schemata (see below) in Scala as a function mapping a sequent to a list of sequents:

```
1 val choose = new ProofRule("choose") {
2   def apply(p: Position) = sq => {
3     val fm = lookup(p, sq)
4     fm match {
5       case Modality(Box, Choose(h1, h2), phi) =>
6         val fm1 = Modality(Box, h1, phi)
7         val fm2 = Modality(Box, h2, phi)
8         val sq1 = replace(p, sq, Binop(And, fm1, fm2))
9         Some((List(sq1),Nil))
10      case _ => None
11    }
12  }
13 }
```

Rule `choose` Line 3 uses `lookup` to access the sequent `sq` at position `p`, and then matches on the shape of that formula. Extension of the rule to make it applicable in other contexts or to other shapes with additional match cases is soundness-critical. This code implements the following sequent calculus proof

rule schemata:

$$([\cup]\wedge\mathrm{L})\ \frac{\Gamma, [\alpha]\phi \wedge [\beta]\phi \vdash \Delta}{\Gamma, [\alpha \cup \beta]\phi \vdash \Delta} \qquad ([\cup]\wedge\mathrm{R})\ \frac{\Gamma \vdash [\alpha]\phi \wedge [\beta]\phi, \Delta}{\Gamma \vdash [\alpha \cup \beta]\phi, \Delta}$$

The loop rule of Fig. 2 is implemented in 61 LOC as a function from the invariant formula to a proof rule, listing how the rule applies in a sequent:

```
1  val loopInduction : Formula => ProofRule =
2  inv => new ProofRule("loopInduction[" + inv + "]") {
3    def apply(pos: Position) = sq => (pos, sq) match {
4      case (RightP(n), Sequent(sig, c, s)) =>
5        val fm = lookup(pos, sq)
6        val initial = replace(pos, sq, inv)
7        fm match {
8          case Modality(Box,Loop(hp, True, inv_hints), phi) =>
9            val inductionstep =
10             Sequent(sig, List(inv),
11                           List(Modality(Box, hp, inv)))
12           val closestep =
13             Sequent(sig, List(inv), List(phi))
14           Some((List(initial, inductionstep, closestep), Nil))
15         case _ => None
16       }
17     .... /* elided lines for diamond modality */
18     case _ => None
19   }
20 }
```

This `loopInduction` rule discards all context in Lines 10–13 to avoid universal closures, which simplifies the implementation considerably but requires users to tediously retain any information needed from the context as part of the loop invariant, which is brittle when editing models. This decision makes loop invariant generation more challenging (not implemented in KeYmaeraD).

KeYmaeraD does not implement differential induction dI of Fig. 2 as a separate proof rule, but instead provides that functionality combined with differential cuts [45, 48, 59] as a `diffStrengthen` rule that augments the evolution domain constraint of a differential equation with a condition `inv`:

```
1  val diffStrengthen : Formula => ProofRule =
2  inv => new ProofRule("diffStrengthen[" + inv + "]") {
3    def apply(pos: Position) = sq => (pos, sq) match {
4      case (RightP(n), Sequent(sig, c, s)) =>
5        val fm = lookup(pos, sq)
6        fm match {
7          case Modality(Box, Evolve(ode,h,inv_hints,sol), p) =>
8            val (ind_asm, ind_cons) = if (Prover.openSet(inv))
9                (List(inv, h),
10                 setClosure(totalDeriv(None, ode, inv)))
11             else (List(h), totalDeriv(None, ode, inv))
12           val inv_hints1 = inv_hints.filter(inv != _)
```

```
13            val fm1 = Modality(Box,
14                               Evolve(ode,
15                                      Binop(And, h, inv),
16                                      inv_hints1,
17                                      sol),
18                               p)
19            val iv = Sequent(sig, h::c, List(inv))
20            val ind = Sequent(sig, ind_asm, List(ind_cons))
21            val str = replace(pos, sq, fm1)
22            Some((List(iv, ind, str), Nil))
23          case _ => None
24        }
25      case _ => None
26    }
27  }
```

This code implements the following sequent calculus proof rules, where $Cl(P)$ is the approximate topological closure of P and P_x^θ substitutes term θ for variable x in P:

$$\text{(dSc)}\ \frac{\Gamma, Q \vdash J, \Delta \quad Q \vdash (J)'^{f(x)}_{x'} \quad \Gamma \vdash [x' = f(x)\,\&\,Q \wedge J]F, \Delta}{\Gamma \vdash [x' = f(x)\,\&\,Q]F, \Delta} \quad \text{(J closed)}$$

$$\text{(dSo)}\ \frac{\Gamma, Q \vdash J, \Delta \quad Q, J \vdash Cl((J)')^{f(x)}_{x'} \quad \Gamma \vdash [x' = f(x)\,\&\,Q \wedge J]F, \Delta}{\Gamma \vdash [x' = f(x)\,\&\,Q]F, \Delta} \quad \text{(J open)}$$

The `diffStrengthen` rule distinguishes between open and closed invariants (Lines 8–11): open differential invariants [45] can assume the invariant in the assumptions `ind_asm` during their inductive proof, on closed invariants only the evolution domain `h` is assumed in `ind_asm` in the induction step, because it would be unsound to assume `inv` [45]. The main implementation in `Prover.totalDeriv` computes the differential invariant condition `ind_cons` for `inv` as a function of the differential equation `ode` and amounts to an extra 1500 lines of Scala code. The result of the `diffStrengthen` rule are three subgoals (Lines 19–22): the invariant `inv` must hold initially from the evolution domain constraint `h` and context `c`; its differential invariant condition `ind_cons` must be preserved from the assumptions `ind_asm`; then `inv` can strengthen the evolution domain constraint `h` in the augmented differential equation (Lines 13–18).

Assignments [:=] are implemented in Scala by turning them into equations:

```
val assign = new ProofRule("assign") {
    def apply(p: Position) = sq =>   {
      val Sequent(sig, c, s) = sq
      val fm = lookup(p, sq)
      fm match {
        case Modality(_, Assign(vs), phi) =>
          var phi1 = phi;
          var sig1 = sig;
          var c1 = c;
          for(v <- vs) v match {
            case (Fn(vr, Nil), tm) =>
```

```
12              val vr1 = Prover.uniqify(vr);
13              phi1 = Prover.renameFn(vr, vr1, phi1);
14              sig1 = sig.get(vr) match {
15                case Some(sg) => sig1.+((vr1, sg))
16                case _        => sig1
17              }
18              val fm1 = Atom(R("=", List(Fn(vr1, Nil), tm)));
19              c1 = c1 ++ List(fm1);
20            ... /* case (Fn(vr, List(arg)), tm) elided */
21          }
22          val sq1 = replace(p, Sequent(sig1, c1, s), phi1)
23          Some((List(sq1), Nil))
24
25        case _ => None
26      }
27    }
28  }
```

This code implements the following sequent calculus proof rule, where y is fresh in the sequent and P_x^y is P with all occurrences of x renamed to y:

$$([:=]\text{eq})\ \frac{\Gamma, y = e \vdash P_x^y, \Delta}{\Gamma \vdash [x := e]P, \Delta}\quad (y \text{ fresh})$$

Rule `assign` does not attempt any simplification (e.g., substitution), but chooses to always rename and introduce equations. It obtains a fresh variable `vr1` with `Prover.uniqify` (Line 12) and then uses `Prover.renameFn` (Line 13) to rename `vr` to `vr1` in the postcondition `phi`. The effect of the assignment is collected as an equation in the context (Lines 18–19), and the signatures of the sequent are updated to include the fresh variable (Lines 14–17).

While the challenge of KeYmaeraD's loop rule is that it discards all information from the context, the challenge of KeYmaeraD's `assign` is the opposite: it retains all assumptions, even assumptions that have become irrelevant or, if substituted, would lead to significantly simpler assumptions. For example, on

$$x = y \vdash [x := x + 1][x := 2x][x := x - 2]x = 2y$$

repeated application of rule `assign` results in

$$x = y, x_0 = x + 1, x_1 = 2x_0, x_2 = x_1 - 2 \vdash x_2 = 2y$$

instead of the simpler $x = y \vdash 2(x + 1) - 2 = 2y$ of $[:=]$ substitution of Fig. 2. Retaining such old versions of variables has a significant impact on the practical performance of real arithmetic [45, Sect. 5.3.2], where just one formula can be the difference between termination within a second and no answer within a day.

Outcomes. The prover KeYmaeraD is an interesting experiment to see what happens when directly implementing a prover to address the shortcomings of

KeYmaera 3, embracing parallel proof search, and generalizing it to distributed hybrid systems. KeYmaeraD is useful for experts familiar with its implementation detail and enables proofs of distributed hybrid systems out of reach for all other verification tools (except later versions of KeYmaera 3). But KeYmaeraD never attracted a sustainable user base. Its blessing of *enabling* low-level control of parallel proof search was simultaneously its curse of *requiring* low-level control to benefit from parallel exploration while simultaneously requiring low-level attention to the logical transformation of formulas. This includes the need for explicit augmentation of invariants with assumptions about constants and tracking of static-single-assignment-renamings of variable generations. But KeYmaeraD is useful for experts who appreciate direct control of proofs close to bare metal.

Statistics. KeYmaeraD comes with 38 built-in proof rules, some of which implement multiple modality cases at once (e.g. `choose`). The statistics give a ballpark estimate even if they are not quite comparable with other provers because KeYmaeraD has only partial support for differential equations and $\langle\cdot\rangle$ modalities and $\exists$ quantifiers, and does not yet provide invariant generation or proof storage.

What Else KeYmaeraD Offers. Most significantly, KeYmaeraD also implements quantified differential dynamic logic QdL for *distributed* hybrid systems [46]. While no interesting hybrid systems were verified with KeYmaeraD, its primary applications use *distributed* hybrid systems technology. KeYmaeraD was used to verify disc-type collision avoidance maneuvers for arbitrarily many aircraft in QdL [29] as well as safety of a surgical robot controller obeying arbitrarily many operating boundaries [25], which are out of scope for all other tools. Since KeYmaeraD is not based on KeY, it did not inherit KeY's automatic proof strategy support and also requires manual instantiation of quantifiers. That is why, ironically, a subsequent implementation of part of the distributed hybrid systems logic QdL in the original KeYmaera 3 prover was more practical.

5 KeYmaera X – An aXiomatic Tactical Theorem Prover With a Small Microkernel

The clean-slate KeYmaera X theorem prover for hybrid systems [20] is a direct implementation of the differential dynamic logic uniform substitution proof calculus [50], complemented by built-in propositional sequent calculus rules for performance reasons, and is implemented in Scala. The uniform substitution style has a significantly simplifying impact on the overall prover design and substantially simplifies soundness arguments thanks to the resulting minimal kernel.

Design Principles. The most important goal in the design of KeYmaera X is its systematic attention to a small-core LCF [30] design, identifying the minimal essential building blocks of a sound hybrid systems prover. An explicit goal of KeYmaera X is to identify a minimal axiomatic prover core (*prover μkernel*)

based upon which a hybrid systems prover can be built easily that is guaranteed to be sound if the μkernel is sound. The crucial ingredient to enable this is dL's uniform substitution calculus [50], which made it possible to avoid proof rule schemata and axiom schemata entirely. Unlike in KeYmaeraD, conceptual simplicity was *not* obtained at the expense of flexibility. Unlike KeYmaera 3, no schematic rule application mechanism such as taclets was used. Uniform substitutions enable a significantly more flexible proof calculus than what KeYmaera 3 had to offer, although never at the expense of soundness. Another goal of the KeYmaera X prover was to increase modularity by having separate responsibilities, e.g., separate modules for prover μkernel, tactic language, tactic libraries, persistence layer, communication layer, web user interface, that are each mostly isolated from one another and, thus, easier to modify or replace separately. For user interaction purposes, KeYmaera X strives for close mnemonic analogy with textbooks, favors proof step result simplicity over internal reasoning simplicity, and presents internal automation steps on demand rather than mixed with user-initiated steps.

Uniform Substitution. The uniform substitution proof rule [50], originally due to Church for first-order logic [14, §35,40], says that if a formula ϕ has a proof, then the result $\sigma(\phi)$ of substituting terms for function symbols, formulas for predicate symbols, hybrid programs for program constant symbols, and context formulas for quantifier symbols, according to uniform substitution σ is proved:

$$(\text{US})\ \frac{\phi}{\sigma(\phi)}$$

The big deal about uniform substitution is that it makes superfluous all schemata and instantiation mechanisms such as taclets and schema variable implementations of KeYmaera 3 and host-language matching code of KeYmaeraD. The soundness-critical part of the implementation of the nondeterministic choice axiom [∪], for example, reduces literally to just providing one concrete dL formula:

$$[a \cup b]P \leftrightarrow [a]P \wedge [b]P$$

This axiom is an ordinary concrete dL formula in which a, b are program constant symbols and P is a nullary quantifier symbol (P can also be thought of as a predicate symbol $p(\bar{x})$ with the vector $\bar{x}$ of all variables as arguments). If a particular instance of this axiom is needed during a proof, all a tactic needs to do is ask rule US to generate an appropriate instance, e.g., with the uniform substitution $\sigma = \{a \mapsto x := x + 1, b \mapsto x' = x, P \mapsto\ x \geq 0\}$ as follows:

$$\text{US}\frac{[a \cup b]P \leftrightarrow [a]P \wedge [b]P}{[x := x + 1 \cup x' = x]\, x \geq 0 \leftrightarrow [x := x + 1]\, x \geq 0 \wedge [x' = x]\, x \geq 0}$$

In KeYmaera X, rule US is mostly used backwards from conclusion to its premise by identifying which substitution σ reduces $\sigma(\phi)$ to a known or easier formula. Unification outside the soundness-critical μkernel finds the appropriate

uniform substitution σ required for a desired inference step (e.g., after matching $[x := x + 1 \cup x' = x]\, x \geq 0$ against $[a \cup b]P$). The same uniform substitution proof rule US resolves, e.g., all the subtleties with the use of assignment axioms where the admissibility conditions of US [50, 51] adequately prevent unsound reasoning. Uniform substitution enables reasoning in context [50], which make uniformly substituted axioms significantly more versatile and flexible than, e.g., the automatic context generalizations of KeYmaera 3, while always rigorously maintaining soundness of such generalizations.

Implementation Realities. Thanks to uniform substitutions, the implementation of axioms in KeYmaera X reduces to providing a direct copy of the dL formula of the axiom given in plain text in the KeYmaera X prover μkernel:

```
Axiom "[++] choice"
  [a;++b;]p(||) <-> [a;]p(||) & [b;]p(||)
End.
Axiom "[:=] assign"
  [x:=f();]p(x) <-> p(f())
End.
Axiom "[:=] assign equality"
  [x:=f();]p(||) <-> \forall x (x=f() -> p(||))
End.
```

These are direct ASCII renditions of the $[\cup]$ choice axiom and the $[:=]$ assignment axiom. The `++` symbol is ASCII for $\cup$ and the notation `p(||)` indicates that `p` is a nullary quantifier symbol (predicational) instead of a nullary predicate symbol as in `p()`. This is all there is to the soundness-critical part in the prover μkernel of the implementation of those axioms, because uniform substitution is available in the μkernel to soundly put concrete terms in for the function symbol `f()`, put concrete formulas in for the predicate symbol `p(.)` of its argument `(.)` or for the predicational symbol `p(||)`, and put concrete hybrid systems in for the program symbols `a;` and `b;`. Thanks to unification outside the soundness-critical μkernel, using such axioms is easily done with a tactic that instructs the unifier to unify with and use the appropriate axiom at the position where it will be applied to:[4]

```
val choiceb = useAt("[++] choice")
```

Tactics get more complicated when different circumstances require different proofs, e.g., assignments sometimes need renaming to avoid conflicts. But tactics are not soundness-critical, so the μkernel is isolated from such complications:

```
@Tactic("[:=]", conclusion = "__[x:=e]p(x)__<->p(e)")
val assignb = anon by { w =>
    useAt("[:=] assign")(w)
  | useAt("[:=] self assign")(w)
  | asgnEq(w)
```

[4] Besides unification, the implementation of the generic `useAt` tactic identifies the (blue) key of an axiom to unify with and generically handles, e.g., equivalence transformations and implicational assumptions that arise during the use of the axiom.

```
}
@Tactic(
    names = "[:=]=",
    premises =   "G, x=e |- P, D",
    //    [:=]=  ------------------
    conclusion = "G |- [x:=e]P, D",
    displayLevel = "all"
  )
val asgnEq = anon by ((w, seq) => seq.sub(w) match {
case Some(Box(Assign(x, t), p)) =>
  val y = freshNamedSymbol(x, seq)
  boundRenaming(x, y)(w) &
  useAt("[:=] assign equality")(w) &
  uniformRenaming(y, x) &
  (if (w.isTopLevel&&w.isSucc) allR(w) & implyR(w) else ident)
})
```

The `assignb` tactic combines axioms and tactics with the `|` combinator to try succinct axioms first and fall back to more complicated tactics only when necessary: it first tries axiom [:=] with `useAt("[:=] assign")(w)` for its succinct result if it is applicable; upon failure, the tactic tries to resolve self-assignments of the form $x := x$, and finally, it applies the generic `asgnEq` tactic that is applicable to any assignment in any context. The `@Tactic` annotations provide naming and rendering information for registering and displaying tactics.

Axioms that are only sound for hybrid systems but not hybrid games use the clunky notation `a{|^@|};` to indicate that `a;` cannot mention the game duality operator ^{d} rendered as `^@` in ASCII so `a;` is a program not a game symbol:

```
Axiom "I induction"
p(||)&[{a{|^@|};}*](p(||)->[a{|^@|};]p(||))->[{a{|^@|};}*]p(||)
End.
```

Thanks to the syntactic internalization of differential operators in differential dynamic logic [50], the implementation of differential invariants is quite simple:

```
Axiom "DI differential invariance"
  ([{c&q(||)}]p(||) <-> [?q(||);]p(||))
                    <- (q(||)->[{c&q(||)}]((p(||))'))
End.
```

A few similarly simple further axioms are needed to, then, remove the resulting differential operator from `p(||)'`, e.g., for distributing differentials over $\wedge$:

```
Axiom "&' derive and"
  (p(||) & q(||))' <-> (p(||))' & (q(||))'
End.
```

But the entire construction is syntactic within the logic dL itself [50] instead of as a soundness-critical algorithm implementing a built-in metaoperator `#DiffInd` with a Java program as in KeYmaera 3 or a Scala program as in KeYmaeraD.

Successes. Probably the biggest success of the KeYmaera X design is how well it has managed to keep the size and complexity of its soundness-critical prover μkernel at only about 2000 lines of code that are mostly straightforward. That makes it much easier to check whether a change could damage the prover, because *i)* soundness is rarely the issue, as code in the μkernel rarely changes, *ii)* completeness is usually the worst that may be affected in most changes, which is more easily noticed with testing (if something no longer proves that did) compared to soundness-critical changes (nobody notices if something suddenly would prove that should not), and *iii)* many changes are monotonic additions that add new tactics whose presence, by design, cannot damage the functioning of the rest of the prover. It is liberating to advance provers under such a design that encourages fast-paced development and teams. The result is a significantly more flexible proof calculus that is able to apply axioms in any binding context inside formulas, checked for soundness by uniform substitution (whereas KeYmaera and KeYmaeraD are restricted to reasoning at top-level operators).

Another major success of the KeYmaera X prover is its versatile user interface [36], which makes the use of the prover transparent compared to the theory. In a nutshell, formal proofs and proof rules can be read and understood equally on paper and in the prover without requiring a shift in perspective.[5] One indication that this design was successful is the fact that undergraduate students are able to learn how to use KeYmaera X in a course [51] well enough even if almost all of them have not had any prior exposure to logic or cyber-physical systems. Contributing in no small part to the successful design of the KeYmaera X user interface is its modular design separate from all the rest of the prover, enabling easier experimentation and complete rewrites.

The modular code base of KeYmaera X is fairly resilient to change. Beyond the intended points of change, it was possible to swap out axioms to prove hybrid games [52], swap out uniform substitution application mechanisms [53], swap out unification, swap out central proof data structures and proof storage, change how formulas in sequents are indexed, and swap out the entire tactic framework.

KeYmaera X has sufficiently modular automatic proof tactics that make it easy to benefit from automatic invariant generators for continuous dynamics such as Pegasus [64]. It is also comparably easy to maintain alternative proof automation tactics in harmonious coexistence, e.g., when it is not clear ahead of time which approach or which combination of approaches will work best.

Challenges. Its focus on soundness makes it comparably easy to spot soundness bugs in KeYmaera X, because its μkernel is short and structurally simple, and because all else gets caught when trying to draw incorrect inferences (e.g., incorrect tactics). But the same cannot be said for performance and completeness bugs. Because KeYmaera X, for performance reasons, does not memorize its proof steps and only reports problems if a proof step was unsound, it is hard to

[5] Ironically, minor notational differences still exist as concessions to ASCII and curly-brace language notation, but major changes such as different proof notations, updates or static-single-assignment-renamed versions of variables are avoided.

notice where tactics found proofs in unnecessarily complicated yet sound ways. For example, when a tactic goes in circles a few times before successfully completing a correct proof, then it is much harder to notice the wasted effort.

Tactic implementation often needs to trade off performance, completeness, and comprehensibility, because it is not only inherently difficult to enumerate all the sound ways of applying axioms in diverse contexts (which is exactly the benefit of using uniform substitution), but also inherently difficult to decide what of that context information should be kept for completeness and which facts to discard to increase comprehensibility and scale. These considerations show up in tactic implementation for KeYmaera X with a performance and completeness impact. But the same challenges surface in KeYmaera and KeYmaeraD where, however, they furthermore cause a soundness impact! For example, consider the following axiom for differential weakening, which allows us to prove postcondition `p(||)` from the evolution domain constraint `q(||)` of an ODE `{c&q(||)}`.

```
Axiom "DW differential weakening"
  [{c&q(||)}]p(||) <-> ([{c&q(||)}](q(||)->p(||)))
End.
```

Axiom `DW differential weakening` by itself does not make the question much easier, since the differential equation `{c&q(||)}` is still around even after applying the axiom from left to right. In order to make progress in the proof, a tactic will need to combine axiom `DW differential weakening` with techniques to abstract the modality away, for example by using axiom `V vacuous` or rule G, as put into operation in the example in Fig. 4.

```
Axiom "V vacuous"
  p() -> [a{|^@|};]p()
End.
```

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

In this example, the proof in Fig. 4b is more concise than Fig. 4a and gives a less surprising result without universal quantifiers. Why may we still want to use the proof technique in Fig. 4a? Consider $x_0 \geq 1 \vdash [x' = 2 \,\&\, x \geq x_0]x \geq 0$, where it is important to retain the initial assumption $x_0 \geq 1$ (rule G is only applicable in an empty context, so would discard assumption $x_0 \geq 1$ by weakening). Many conjectures are best served (for users and automation) by concise reasoning that leaves less clutter around, but for completeness it becomes necessary to retain certain facts. A prover has to decide what is appropriate in which situation.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash \forall x\,(x \geq 0 \to x \geq 0)}\,\text{QE}}{\vdash [x' = 2 \,\&\, x \geq 0]\forall x\,(x \geq 0 \to x \geq 0)}\,\text{V}}{\vdash [x' = 2 \,\&\, x \geq 0](x \geq 0 \to x \geq 0)}\,\forall^{\alpha}}{\vdash [x' = 2 \,\&\, x \geq 0]x \geq 0}\,\text{DW}$$

(a) Abstraction with axiom V vacuous

$$\dfrac{\dfrac{\dfrac{*}{\vdash x \geq 0 \to x \geq 0}\,\text{QE}}{\vdash [x' = 2 \,\&\, x \geq 0](x \geq 0 \to x \geq 0)}\,\text{G}}{\vdash [x' = 2 \,\&\, x \geq 0]x \geq 0}\,\text{DW}$$

(b) Abstraction with rule G

Fig. 4: Tactic alternatives for differential weakening by vacuity or generalization

Learning from the successes of KeYmaeraD, initial attempts on a tactic framework for KeYmaera X emphasized concurrent and speculative tactic execution in the spirit of modern processor designs for performance reasons. Relying only on aggressively concurrent execution turns out to be a misguided idea during development, because, even with sound proofs, concurrent execution makes it near impossible to debug where and why tactics are failing if there is no reliable way of triggering the same failure in the same order again. The current sequential tactic framework entirely dismisses all aspirations for concurrent execution (except when using explicit parallel tactic combinators), and is significantly more successful in practice.

For modularity reasons, KeYmaera X has a tactic interpreter that runs a tactic on a sequent to completion returning the remaining open premises (if any, else the sequent is proved). This design results in a clear separation of concerns compared to the rest of the prover. But, unlike in KeYmaera 3, it is difficult to meaningfully interrupt an automatic proof in the middle and then help out interactively. This is related to the fact that, other than an exhaustive list of all individual proof steps, there is no reliable way to record the resulting proof, because there is no guarantee that interrupting the automatic tactic after the same amount of time or the same number of inferences in the future will result in the same remaining open premises. To mitigate, KeYmaera X provides ways of running tactics in exploratory mode, whether the tactic completes successfully or not, and a second tactic interpreter records and exposes internal proof steps.

The reliance of the KeYmaera X user interface on a web browser makes it much easier to distribute proof development to multiple users and provides browser-powered rendering capabilities. But it requires dealing with the acute idiosyncracies of browser dependencies, unreliability of JavaScript and its libraries.

Outcomes. Fairly subtle properties of major complicated systems have been successfully verified with KeYmaera X, including the Next-generation Airborne Collision Avoidance Systems ACAS X [24], obstacle avoidance [33] and waypoint navigation [13] for ground robots, and air pressure brakes for trains [31]. The syntactic rendition of proofs as tactics in KeYmaera X was also exploited to enable provably safe reinforcement learning in cyber-physical systems [21], as well as tactical verification of component-based hybrid systems models [40]. ModelPlex [37] crucially exploits the axiomatic approach of KeYmaera X that enables in-context reasoning and avoids splitting ModelPlex proofs into multiple branches, so that all runtime conditions are collected in a single proof goal. The first implementation of ModelPlex in KeYmaera [35] with its restriction to top-level rules branches heavily, which requires a soundness-critical step of collecting and factoring all open branches from an unfinished proof. The strict separation of generating invariants, solutions of differential equations, and other proof hints, from tactically checking them enables KeYmaera X to include numeric and other potentially unsound methods (e.g., for generating barrier certificates) into its invariant generation framework Pegasus [64] for nonlinear ODEs.

These case studies confirm the advanced scale at which differential dynamic logic proving helps make hybrid systems correct even for fairly complex applications with very subtle properties. The use of tactics in the Bellerophon tactic language of KeYmaera X [19] played a big role in proving them. When beyond the reach of full automation, proof tactics are significantly easier to rerun, modify, and check compared to point-and-click interactive proofs. The comparably fast-paced development with a small μkernel underneath a large library of tactics also makes it easier to advance proof automation, including automation of complete differential equation invariant proving [59]. The curse of improving proof automation, however, is that tactics may have to be adapted when more of a proof completes fully automatically and the remaining formulas change.

The ARCH competition [38] highlights significant improvements for full automation of continuous dynamics from KeYmaera to KeYmaera X, but also that the tactics in KeYmaera X, while more automatic, still lose efficiency compared to the proof search and checking procedures of KeYmaera. The benefit of separating proof automation in tactics from the μkernel manifests in performance and automation improvements between KeYmaera X versions: proof automation in KeYmaera X improved [38] to the level of scripted proving reported a year earlier [39], and additional scripted functionality became available.

Statistics. Besides uniform substitution, renaming, and propositional sequent calculus rules with Skolemization, the KeYmaera X μkernel provides 5 axiomatic proof rules with concrete dL formulas and 54 concrete dL formulas as axioms. A large number of tactics (about 400) are built on top of this μkernel and are, thus, not soundness-critical.

What Else KeYmaera X Offers. Beyond serving as a hybrid systems theorem prover, KeYmaera X implements differential game logic for hybrid games [49], which, quite unlike the case of games added to KeYmaera 3 [61], required only a minor change. This is an example illustrating why the axiomatic approach makes it significantly easier to soundly change the capabilities of a prover. The axiomatic approach is also beneficial to obtain formal guarantees about the prover μkernel itself [11], and serves as the basis for transforming hybrid systems proofs to verified machine code through the compilation pipeline VeriPhy [12].

6 Comparison of Underlying Reasoning Principles

In this section, we compare how differences in the underlying reasoning principles result in differences in the concrete mechanics of conducting sequent proofs and differences in the code base organization.

Implementation Comparison By Example. We discuss assignments, loops, and differential equations as illustrative examples of the implementation choices across KeYmaera, KeYmaeraD, and KeYmaera X.

Assignments. Assignments, even though seemingly simple at first glance, become tricky depending on the binding structure of the *imperative* programs that follow. Let us briefly recap how handling assignments differs across provers:

- KeYmaera turns all assignments into updates, but delegates the soundness-critical task of how to apply these updates to the update simplifier and leads to soundness-critical decisions of invisibly leaving updates around all taclets;
- KeYmaeraD opts for the always-safe choice of introducing universal quantifiers and turning all assignments into extra equations, but this results in a plethora of similar symbols and equation chains that can (i) be confusing for users, (ii) be exceedingly challenging for real arithmetic solvers, and (iii) increase the complexity of identifying loop invariants and ODE invariants;
- KeYmaera X exploits the safety net of the underlying uniform substitution algorithm: its tactic replaces the free occurrences of the assigned variable, and only if the prover kernel rejects this[6], falls back to introducing equations.

The following example illustrates the difference in the behavior between KeYmaera (**K3** for short below), KeYmaeraD (**KD**), and KeYmaera X (**KX**):

$$\dfrac{\dfrac{\dfrac{}{x=1 \vdash \{x:=x+1\}[?x\geq 1]x\geq 0}\,\textbf{K3}\ (\text{update})}{x=1, x_1 = x+1 \vdash [?x_1\geq 1]x_1\geq 0}\,\textbf{KD}\ (\text{equation})}{x=1 \vdash [?x+1\geq 1]2\geq 0}\,\textbf{KX}\ (\text{substitution})}{}[:=]\ x=1 \vdash [x:=x+1][?x\geq 1]x\geq 0$$

In this example, the assigned variable x occurs free but not bound in the formula $[?x \geq 1]x \geq 0$ following the assignment and so can be substituted. Despite this, KeYmaera and KeYmaeraD stick to their fixed behavior of introducing updates and equations, respectively. KeYmaera X uses a substitution in the above example, and adapts its behavior appropriately using free and (must)bound variables from the static semantics of dL [50]. The wide variety in the examples below of how to best handle assignments in sound ways explains why that is best decided outside the soundness-critical prover core.

Must-bound not free: After the assignment, the assigned variable is definitely bound but not free, so the assignment has no effect:

$$\dfrac{\vdash [x:=3]x\geq 3}{[:=]\ \vdash [x:=2][x:=3]x\geq 3}$$

Free and must-bound: After the assignment, the assigned variable is free and definitely bound, so all free occurrences can be substituted:

$$\dfrac{\vdash [x:=2+1]x\geq 3}{[:=]\ \vdash [x:=2][x:=x+1]x\geq 3}$$

[6] Earlier implementations of the KeYmaera X assignment tactic attempted to syntactically analyze the formula to decide which axiom to use, which is essentially the task of the update simplifier in KeYmaera. This approach turned out to be too error-prone, unless the tactic exactly mimics the uniform substitution algorithm.

Free and maybe-bound: The assigned variable is bound on some but not all paths of all runs of the program, so not all free occurrences are replaceable and therefore KeYmaera X introduces an equation (for traceability KeYmaera X retains original names for the "most recent" variable and renames old versions in the context; higher index indicates more recent history):

$$[:=]\frac{x_0 = 1, x = 2 \vdash [\{x' = x\}]x \geq 2}{x = 1 \vdash [x := 2][\{x' = x\}]x \geq 2}$$

Free and must-bound before maybe-bound: The assigned variable is definitely bound again later, so the free occurrences can be replaced:

$$[:=]\frac{\vdash [x := 2 + 1 \cup x := 3][(x := x + 1)^*]x \geq 3}{\vdash [x := 2][x := x + 1 \cup x := 3][(x := x + 1)^*]x \geq 3}$$

In context substitutable: The assignment occurs in the context of a formula and substitution is applicable:

$$[:=]\frac{\vdash [\{x' = -x^2\}](x + 1)^2 \geq 2}{\vdash [\{x' = -x^2\}][x := x + 1]x^2 \geq 2}$$

In context not substitutable: The assignment occurs in the context of a formula but substitution is not applicable because the assigned variable is maybe bound later:

$$[:=]\frac{\vdash [(x := x + 1)^*]\forall x\,(x = 2 \rightarrow [\{x' = 3\}]x \geq 2)}{\vdash [(x := x + 1)^*][x := 2][\{x' = 3\}]x \geq 2}$$

Right-hand side may be bound: The right-hand side of the assignment cannot simply be substituted in because it is maybe bound on some paths:

$$[:=]\frac{y < 0, x = y^2 \vdash [y := -x \cup \{y' = x\}]y < 0}{y < 0 \vdash [x := y^2][y := -x \cup \{y' = x\}]y < 0}$$

Note that a must-bound occurrence before any maybe bound occurrences again enables plain substitution, as shown below:

$$[:=]\frac{y < 0 \vdash [x := y^2 + 1][y := -x \cup \{y' = x\}]y < 0}{y < 0 \vdash [x := y^2][x := x + 1][y := -x \cup \{y' = x\}]y < 0}$$

The benefits and drawbacks of implementations are summarized in Table 1.

Loop Induction. Loop induction showcases how the specific implementations in the provers result in different demands on user intervention afterwards. KeYmaera favors completeness and retains all context in the taclet but requires users to discard unwanted assumptions (distracting users and arithmetic procedures [45]). KeYmaeraD implements the obvious sound rule of removing all context but requires users to explicitly retain any desired assumptions in the loop invariant

Table 1: Assignment comparison

	Pros	Cons
KeYmaera	Common proof step result (updates)	Complexity hidden in critical update simplifier, affects other taclets
KeYmaeraD	Common proof step result (equations)	Creates distracting equations and Skolem symbols (challenging for users and QE and invariants)
KeYmaera X	Favors simplicity, creates variables and quantifiers only when necessary, works in context	Less easily predictable proof step result makes automated follow-up tactics challenging

(which needs fragile adaptations when the model or proof changes). The KeYmaera X tactic attempts to strike a balance between the two in the usual cases, requiring users to retain extra assumptions in unusual cases. KeYmaera X tactics enable additional features, such as referring to the state at the beginning of the loop with the special function symbol *old*. The example in Fig. 5 illustrates the differences in loop induction implementations (the loop invariant $x \geq 1$ for KeYmaeraD needs to be augmented with the additional constant fact $\ldots \wedge b > 0$). Completeness of the loop tactic is more challenging than for assignment, since

$$\textbf{K3}\quad x=1, b>0 \vdash x \geq 1$$
$$\textbf{KD}\quad x=1, b>0 \vdash x \geq 1 \wedge b > 0$$
$$\textbf{KX}\quad x=1, b>0 \vdash x \geq 1$$

(a) Base case

$$\textbf{K3}\quad x=1, b>0 \vdash \forall x\,(x \geq 1 \rightarrow x \geq 0)$$
$$\textbf{KD}\quad x \geq 1 \wedge b > 0 \vdash x \geq 0$$
$$\textbf{KX}\quad b>0, x \geq 1 \vdash x \geq 0$$

(b) Use case

$$\textbf{K3}\quad x=1, b>0 \vdash \forall x\,(x \geq 1 \rightarrow [x := x + 1/b]x \geq 1)$$
$$\textbf{KD}\quad x \geq 1 \wedge b > 0 \vdash [x := x + 1/b](x \geq 1 \wedge b > 0)$$
$$\textbf{KX}\quad b>0, x \geq 1 \vdash [x := x + 1/b]x \geq 1$$

(c) Induction step

$$\text{loop}\frac{\text{Base case (5a)} \qquad \text{Use case (5b)} \qquad \text{Induction step (5c)}}{x=1, b>0 \vdash [(x := x + 1/b)^*]x \geq 0}$$

Fig. 5: Difference in loop induction: KeYmaera (**K3**), KeYmaeraD (**KD**), and KeYmaera X (**KX**)

it has to transform sequents into the shapes expected by the axioms used internally. An early version of the tactic lost constant facts when they were not isolated or when they were "hidden" in negated form in the succedent (e.g., lost $b > 0$ in a sequent $x = 1 \wedge b > 0 \vdash [(x := x + 1/b)^*]x \geq 1$). The current tactic

Table 2: Loop induction comparison

	Pros	Cons
KeYmaera	Fast one step rule, common proof step result (keeps full context, universal closure for soundness)	Universal closure with new names, needs manual removal of undesired assumptions, hard to extend features (careful: soundness)
KeYmaeraD	Fast one step rule, common proof step result (discards context)	Needs manual action to retain necessary assumptions, bad for invariant generation, hard to extend features (careful: soundness)
KeYmaera X	Less user intervention in the usual cases, easily extensible (e.g., *old* terms for ghosts)	Completeness is challenging in unusual cases, users may not immediately be able to help since usual cases work on their own

applies α-rules first to attempt isolating constant facts, so it still loses information, e.g., when nested inside $(x \geq 1 \land b > 0) \lor (x \geq 4 \land -b < 0)$. The benefits and drawbacks of implementations are summarized in Table 2.

Differential Induction. Similar to loop induction, KeYmaera favors completeness and retains all context. KeYmaeraD implements differential invariants not as a separate rule, but integrated with differential cuts in a single "differential strengthen" rule. KeYmaera X is closer to KeYmaera, but automatically closes the resulting goals if not explicitly asked to stop at an intermediate result. The following example illustrates the difference between the differential induction implementations (✓ indicates when goals are closed automatically).

$$\begin{array}{lll} & \text{(init)} & \text{(step)} \\ \textbf{K3} & x^2+y^2=1 \vdash x^2+y^2=1 & x^2+y^2=1 \vdash \forall x \forall y\,(2xy+2y(-x)=0) \\ \textbf{KD} & x^2+y^2=1 \vdash x^2+y^2=1 & \vdash 2xy+2y(-x)=0 \\ \textbf{KX} & x^2+y^2=1 \vdash x^2+y^2=1 \;✓ & x_0^2+y_0^2=1 \vdash [x':=y][y':=-x]2xx'+2yy'=0 \;✓ \\ \hline \text{DI} & x^2+y^2=1 \vdash [x'=y,y'=-x]x^2+y^2=1 & \end{array}$$

Solving ODEs. The intuitively (but not computationally!) easiest way of proving a property of a differential equation $[x'=f(x)\,\&\,Q]P$ is to replace it with a property of its solution [43, 48, 50] with a universal quantifier for all times $t \geq 0$. When $y(t)$ is the solution over time t of the above differential equation (and other side conditions hold [43, 48, 50]), then $[x'=f(x)\,\&\,Q]P$ is equivalent to:

$$\forall t{\geq}0 \left((\forall 0{\leq}s{\leq}t\,[x:=y(s)]Q) \rightarrow [x:=y(t)]P\right)$$

The inner quantifier checks that the evolution domain constraint Q was true at every intermediate time s. It would be correct to prove $[x'=f(x)\,\&\,Q]P$ by

proving a formula that merely assumes that Q was true at the end time t:

$$\forall t{\geq}0 \left([x := y(t)]Q \rightarrow [x := y(t)]P\right)$$

Often it is more efficient to just consider the endpoint, but sometimes completeness requires the presence of the assumption about all intermediate times s. That is why all three provers implement both versions. Of course, either reasoning principle is only correct when the side conditions hold [43, 43, 48], most importantly that $y(t)$ actually is a solution of the differential equation (system) $x' = f(x)$ and satisfies the symbolic initial value $y(0) = x$.

KeYmaera trusts the differential equation solver of Mathematica or the Orbital[7] library to produce correct solutions (the conversions and taclet infrastructure are about 2k lines of code). KeYmaeraD trusts its builtin integrator and linear algebra tools (about 1k lines of code), or the user to annotate the correct solution of the differential equation with `@solution` in the model. KeYmaera X implements differential equation solving purely by proofs in tactics (about 1.5k lines of code) based on one axiom for solving constant differential equations [50]:

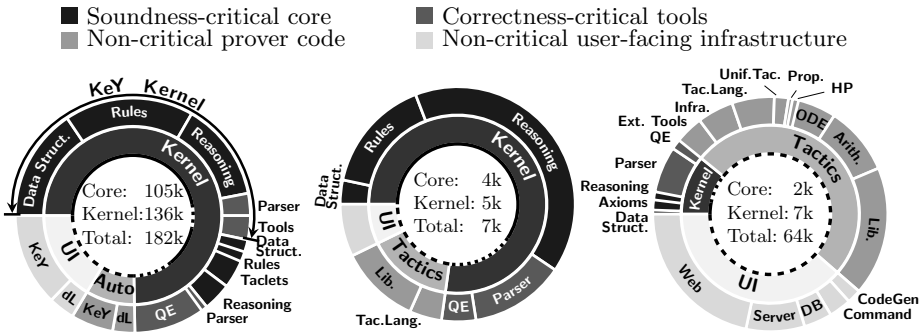
```
Axiom "DS& differential equation solution"
  [{x'=c()&q(x)}]p(|x'|) <-> \forall t (t>=0 ->
  \forall s (0<=s&s<=t->q(x+c()*s)) -> [x:=x+c()*t;]p(|x'|))
End.
```

Code Structure and Soundness-Critical Proof Infrastructure. Fig. 6 summarizes the code base of the KeYmaera family provers, structured as follows:

Kernel core data structures and soundness-critical rules, parsing and printing, as well as interfacing and interaction with external tools for arithmetic;
Tactics proof primitives, and framework support for automation and scripting;
UI non-critical user-facing infrastructure and proof presentation.

KeYmaera (Fig. 6a). The KeY core provides data structures to represent sequent proofs, express logics and specification languages, as well as soundness-critical support for taclet implementation and soundness-critical reasoning infrastructure to conduct proofs and analyze cases. The KeYmaera core extends the KeY core with data structures for dL, dL rules and taclets (see Table 3 for details), as well as reasoning support for dL (formula analysis, computing transition models of hybrid programs, and image computation). The KeY and KeYmaera parsers are not strictly soundness-critical, since proofs are rerun from scratch from pretty-printed input models. KeYmaera interfaces with numerous external tools for flexible QE support, which results in a considerably larger QE package than KeYmaeraD and KeYmaera X. The taclet mechanism registers taclets with proof automation strategies, as well as with UI elements (taclets appear automatically in context menus, and create dialog boxes and input elements).

[7] The Orbital library is a Java library providing object-oriented representations and algorithms for logic, mathematics, and computer science.



(a) KeYmaera: extends existing dynamic logic prover KeY (extensions critical)

(b) KeYmaeraD: direct impl. in host language (extensions critical)

(c) KeYmaera X: small uniform substitution core (tactic extensions non-critical)

Fig. 6: Source code structure (black/solid arc: soundness-critical core, dark gray/dotted arc: correctness-critical tools, medium gray/dashed arc: non-critical prover code, light gray/dashed arc: non-critical user-facing infrastructure),

KeYmaeraD (Fig. 6b). The KeYmaeraD core includes data structures to represent the dL syntax, sequents, proof trees, and rationals. Differential dynamic logic is implemented entirely with soundness-critical builtin rules (for details see Table 3), with reasoning support to apply rules, simplify arithmetic, schedule reasoning jobs, and propagate results through the proof tree. The KeYmaeraD parser is not strictly soundness-critical, since proofs cannot be stored, but are always run from scratch. The QE package is restricted to Mathematica. The tactic language provides basic combinators and maps dL operators to builtin rules; the library contains a selection of basic automation procedures (e.g., exhaustively apply α/β-rules, hybrid program simplifications). The UI renders the proof tree directly to a scrollable Java JTree but does not allow interaction with the proof.

KeYmaera X (Fig. 6c). The KeYmaera X core includes data structures for the syntax, nonschematic axioms and rules (see Table 3) together with managing sequents, and core reasoning functionality to compute static semantics, uniform substitution, and renaming. Its parser is not strictly soundness-critical, since it is safeguarded with cryptographic checksums (or print-reparse-checks) on storage and users can inspect printed outputs. The QE functionality provides transformations to and from data structures of external solvers and the interaction with those solvers. The tactics framework includes additional external non-critical tools for invariant generation, simplification, and counterexample generation. It provides non-critical but convenient proof infrastructure, such as unification, expression traversal, combined renaming and substitution. The tactic language allows users to compose/write new tactics and provides tactic interpreters, ways to store tactic results as lemmas, and to interface with external tools. Noteworthy tactics packages are Unif. Tac. to apply axioms by unification, Prop. for

propositional reasoning, HP for hybrid programs, ODE for substantial differential equations proof automation, and Arith. for arithmetic, equality rewriting, quantifier instantiation/Skolemization, interval arithmetic, and simplification. The tactic library bundles those with a library of lemmas derived from the core axioms, and provides additional reasoning styles, such as ModelPlex [37], component-based proofs, and invariant provers for loop/ODE invariant search.

Comparison of Core Taclets, Rules, and Axioms. Table 3 compares the size of the core dedicated to expressing differential dynamic logic: KeYmaera taclets are slightly more verbose[8] than KeYmaeraD host-language rule implementations and KeYmaera X axioms[9]. The most noteworthy difference in the code structure is how soundness-critical code is scattered across the code base. Both KeYmaera and KeYmaeraD have soundness-critical code in the core as well as in the tactics, while, overall, only about a third of their code bases are non-critical. KeYmaera X, in contrast, confines soundness-critical code entirely to the core data structures, uniform substitution, bound renaming, and the small set of builtin rules. The provers differ in their proof manipulation: in KeYmaera the proof tree data structure keeps track of proof steps and open goals, in KeYmaeraD the tactic framework with its proof tree is responsible for correctly combining proof steps into a proof, whereas in KeYmaera X only the core can manipulate proof objects and a proof is obtained by transforming the original proof object containing the conjecture into one with an empty list of subgoals. KeYmaera X, thus, is the only LCF-style prover among the three.

Table 3: Core size LOC: KeYmaera taclets, KeYmaeraD rules, KeYmaera X axioms

	K3	**KD**	**KX**
Propositional	140	257	212
HP	1202	352	55
ODE	322	241	122
Arithmetic	1033	103	0

KeYmaera: +39k LOC rule code

Code Size. In terms of overall size, KeYmaeraD has by far the smallest overall code base, but, as a bare-bones prover, comes without the convenient user interfaces and proof support and automation of KeYmaera and KeYmaera X. More importantly, however, KeYmaera X comes with the smallest and least complex soundness-critical core, which has direct consequences for the trustworthiness and the extensibility of the prover. Extension of KeYmaera requires adding new soundness-critical taclets and taclet support code, and registering those with the taclet application mechanism. It is often impossible to extend KeYmaera without making changes to soundness-critical code. Extension of KeYmaeraD typically requires adding new soundness-critical rules in the prover core in the host language, while the primary purpose of the tactics framework is to express problem-specific proof scripts; tactic registration is only necessary to add tactics

[8] The main taclet code complexity, however, is hidden in the soundness-critical implementation code that is backing the taclets.

[9] KeYmaera X, in addition to axioms, uses host-language rule implementations for propositional rules, which are included in the count.

to automation, since proofs are expressed in the host language. Extension of KeYmaera X typically requires adding new tactics, and registering those with tactic automation if they should be used automatically; new soundness-critical axioms are only necessary when extending the underlying logic. Another noteworthy difference is in the sizes of QE packages: KeYmaera dedicates considerable code size to interfacing with numerous external tools; KeYmaeraD focuses on only Mathematica; KeYmaera X interfaces with Z3 and Mathematica, but its architecture of reproving inputs of external tools also enables separating critical QE calls from non-critical invariant and counterexample generation.

Summary. The code bases differ considerably in their size, the way they separate soundness-critical from non-critical reasoning, and the way they support extension; the total size of KeYmaeraD may be small enough to justify the use of soundness-critical builtin rules, but with increasing size at or beyond the size of KeYmaera, it becomes increasingly difficult to justify the correctness of extensions. Separation in the style of KeYmaera X enables extensibility and courageous automation in tactics. The considerable amount of code that both KeYmaera and KeYmaera X dedicate to the user interface and proof support results in quite different user interaction experience, as discussed next.

7 User Interaction

The presentation of sequent proofs on limited screen estate is challenging, and the design choices of which information to readily emphasize and which information to make available on request considerably influence the user interaction.

KeYmaera. For user interaction, KeYmaera emphasizes the tree structure of sequent proofs and internal automation steps when presenting proof obligations. Proof obligations and formulas are rendered almost in ASCII syntax, see Fig. 7.

Users perform proofs by interacting exclusively on the top-level operators by selecting proof steps from a context menu. A proof tree is beneficial for displaying the history and source of proof obligations (which steps produced a certain subgoal), but can be challenging to immediately spot the open proof obligations, especially those resulting from automation. In KeYmaera, proof automation adds its internal steps to the proof tree, which is useful to interact with automation, learn doing proofs by observing proof automation on simple examples, but can be hard for users to map with their interaction (a single click produces many steps, where did automation start, which subgoals were there already, which ones are new). Except for experienced users, it's also hard to map nodes in the tree to the input formula and statements in the input program, which, however, is vital information for users to understand the open goal when their input is required. The results of intermediate and internal steps are non-persistent, which makes step-by-step interaction faster but loading and continuing unfinished proofs slow (all proof steps need to be redone on load). Table 4 summarizes the benefits and drawbacks of the KeYmaera proof tree presentation, its top-level interaction, and its presentation of automation internals.

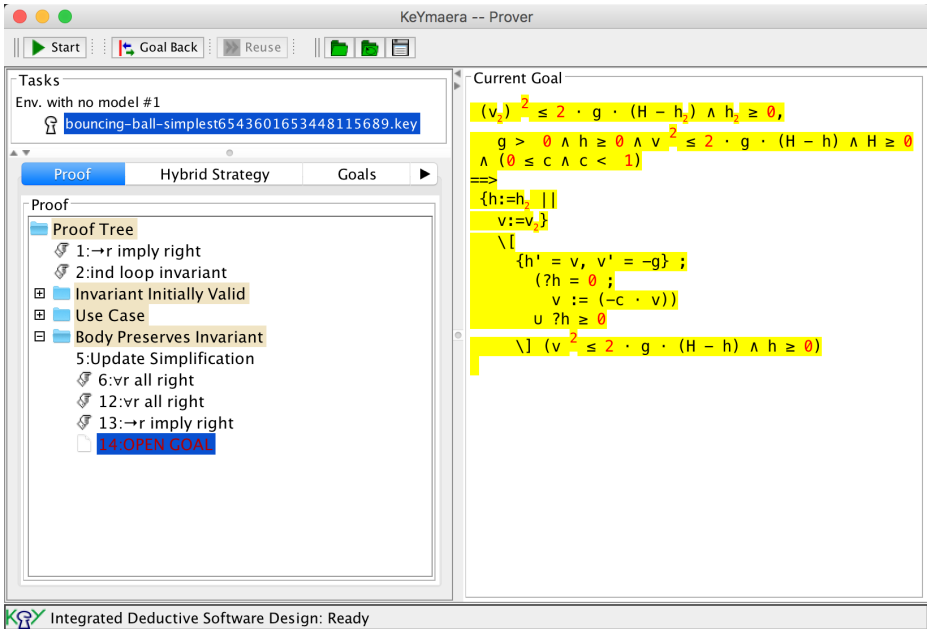


Fig. 7: KeYmaera UI: proof tree left, sequent of selected node right

Table 4: KeYmaera user interaction summary

	Pros	Cons
Proof tree	History and source of proof obligations obvious	Dissimilar to textbook proofs, spotting open obligations after automation challenging
Top-level interaction	Next proof step often obvious, unclear if on formula or update	Duplicate proof effort after branching
Automation internals	Learn proving by observing, more robust on replay	Challenging to undo
Automation settings	Fine-grained automation steering	Settings are part of proof and lost if different per branch
Non-persistent intermediate steps	Faster on performing steps	Slow on proof loading

KeYmaeraD. Proofs in KeYmaeraD are explicitly programmed in Scala, either as a Scala program or in Scala's native read-evaluate-print loop and displayed as a listed tree in its user interface (Fig. 8). In dL, proof steps primarily manipulate programs and formulas, but branch only occasionally; rendering such deep trees verbatim may waste screen estate and require cumbersome horizontal and vertical scrolling simultaneously to navigate a proof. KeYmaeraD comes with a

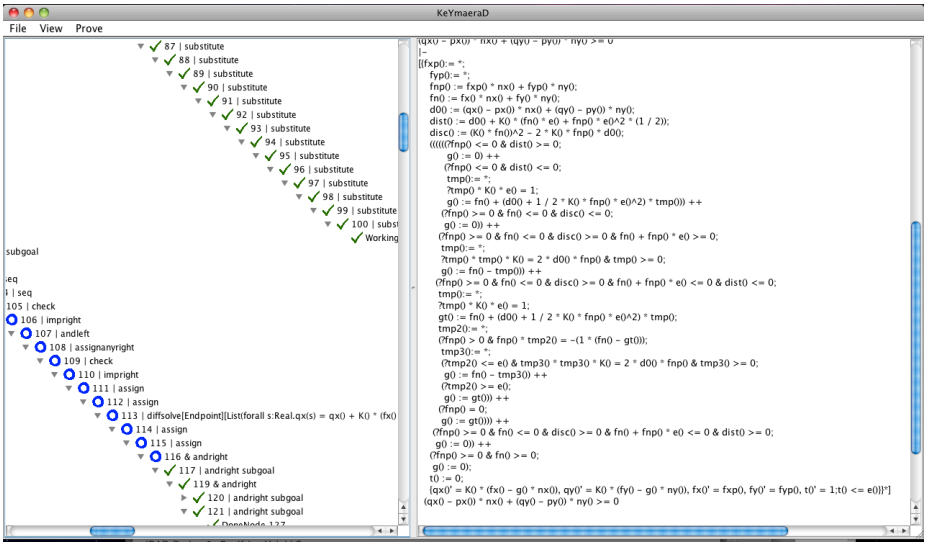


Fig. 8: KeYmaeraD UI: proof tree left, sequent of selected node right

Table 5: KeYmaeraD user interaction summary

	Pros	Cons
Proofs in host language	Flexible, readily available development tools	Hard for novice users
AND/OR proof tree	Explore alternatives, history obvious	Dissimilar to textbook sequent proofs, spotting open goal combinations challenging
Top-level interaction	Next proof step often obvious	Duplicate proof effort after branching

small library of base tactics corresponding to each proof rule and provides tactic combinators to express proof alternatives, repeat proof steps, compose proof steps, branch the proof, or try rule applications. Expert users can leverage the full flexibility of the programming language and its development tools when implementing proofs and tactics, but novice users get little help for getting started. Table 5 summarizes the benefits and drawbacks of implementing proofs in a host language on an AND/OR tree with only top-level user interaction in sequents.

KeYmaera X. KeYmaera X emphasizes open proof tasks and close mnemonic analogy with textbooks and user inputs both in its presentation as a HTML rendering of sequent proofs and in its user interaction, see Fig. 9. The client-side JavaScript-based web UI enables flexibility in rendering proofs and is operating-system independent, but at the cost of additional data structure serialization (not necessary in a native UI integrated with the prover) and handling browser

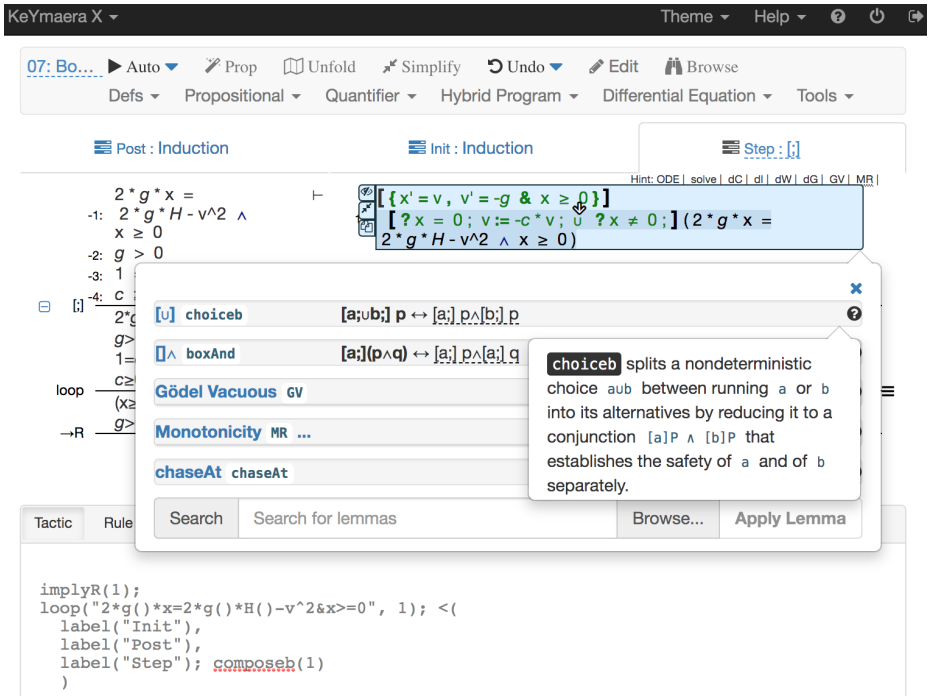


Fig. 9: KeYmaera X UI: Proof menu, open goals in tabs with full sequent proof deduction, proof hints and help in context menu, recorded tactic at the bottom

idiosyncrasies. The user interface design strives for a familiar look-and-feel with tutoring for novice users, flexibility to accommodate various user preferences in reasoning styles, traceability of internal automation steps, and experimentation with custom proof strategies [36]. The internal steps of automation and tactics are hidden from users and expanded only on demand. Open goals are represented in tabs on the UI, which, in contrast to a proof tree, emphasizes the remaining proof agenda. Each tab shows an open goal as the topmost sequent, and renders the history of proof steps as deduction paths from the proof tree root to that open goal, and so the proof tree is implicit (for expert users, however, an explicit proof tree can be beneficial to inspect the structure of the proof [22]). Tab headings present an opportunity to highlight short summaries about the open goal and the proof history, but require careful ordering and—just like nodes in a proof tree—need visual cues to guide user attention (e.g., draw attention to tabs with counterexamples). Users perform proof steps at any level even in the context of other formulas and programs, which helps reduce duplicate proof effort for experienced users. Flexible reasoning at any level may help users follow their natural thought process but may make it harder to discover all applicable proof steps (which is tackled with redundant information about proof steps in context menus, proof menus, proof hints, and searchable lists of all available tactics).

Table 6: KeYmaera X user interaction summary

	Pros	Cons
Proof obligations	Focus on proof tasks	Proof history only shown after mouse click
Sequent proof rendering	Close to textbook sequent proofs	Separate HTML rendering
Explicit on-demand automation internals	Presentation corresponds with user interaction	Recomputes internal steps, tactic changes may fail replay
Interaction anywhere	Reduce proof effort duplication, follow natural thought process, beneficial for efficiency and responsiveness	Hard to anticipate all potential uses of tactics, discoverability (menu, proof hints, context menu)
Persistent intermediate steps	Fast on proof loading	Slower on step-by-step interaction

The implementation of in-context proof step application and custom user proof search strategies crucially requires the safety net of uniform substitution, since it is otherwise challenging to anticipate all potential tactic uses and safeguard them to ensure only sound syntactic transformations in sound contexts. Table 6 summarizes the benefits and drawbacks of focusing on presenting proof obligations in favor of a proof tree, sequent proof rendering, interaction anywhere, and persistent intermediate steps.

8 Related Work

The development history from LCF to Isabelle/HOL is explained by Paulson *et al.* [41] with insights into the design choices and evolution of proof requirements that drove the development of the tool ecosystem.

The literature [68, 69] compares provers in terms of their library size, logic strength, and automation on mathematical problems (with references for several other prover comparisons between Isabelle/Coq/NuPRL/HOL/ALF/PVS). A qualitative comparison of the Isabelle, Theorema, Mizar, and Hets/CASL/TPTP theorem provers [26] highlights user-facing differences in expressiveness, efficiency, proof development and management, library coverage, documentation, comprehensibility, and trustworthiness. Comparisons of the Mizar and Isar proof languages [67] focus on a technical perspective and user experience.

Of complementary interest is the comparison of performance and number of problems solved in theorem prover competitions [4, 65] and hybrid systems tool competitions [17]. While those are valuable from a utilitarian perspective, we argue that bigger insights are to be had from looking inside the box to understand the technical consequences of prover design decisions and, more importantly, how one best enables prover performance while preventing to accidentally "solve" a problem incorrectly due to a soundness mistake in the implementation. Arguably

the biggest level of soundness assurance can be had from formal verification of the prover, which is possible [11] thanks to the simplicity of the uniform substitution calculus of dL. While solving all soundness challenges in principle, this does not yet lead to a high-performance prover kernel implementation in practice.

9 Takeaway Messages

Advantages of KeYmaera 3 implementation approach: very easy to get started with minimal effort (in large part due to the KeY prover basis) and without much thought about what goes where, taclets minimize the need to separate a rule from strategic advice on how to apply it. A gigantic advantage, although specific to the particular KeY basis, is that prover development gets proof visualization and interaction features from day one, which is an immensely helpful debugging aid for rule implementations, proof automation, and the conduct of case studies. The ability to interrupt proof automation and roll it back partially before interacting and handing off to automation again also leads to fairly powerful proof capabilities for experts at the expense of a loss in robustness and traceability compared to the modular tactical proofs of KeYmaera X. Advanced proof strategies or proof scripts, however, also become prohibitively complicated.

Advantages of KeYmaeraD implementation approach: extremely transparent with all details in one place about what happens. A downside is that there is virtually no isolation of soundness-critical core versus the rest of the prover. Adding automation or writing proofs requires detailed knowledge of the internal prover implementation details. KeYmaeraD was successful in terms of its parallel distributed proof search. The reason why the curse of concurrency was significantly less of an issue for KeYmaeraD than KeYmaera X is that KeYmaeraD does not optimize what reasoning to use for which formula, but only ever has one reasoning style for every connective. This hints at a tradeoff that parallel proofs become pleasant only when the individual subproofs become suboptimal and when sufficient computing resources for a simultaneous exploration of all options are available. This comparison is not exhaustive, however, because KeYmaeraD does not provide complex tactics such as differential equation solving [50], differential invariant axiomatization [59], or proof-based invariant generation [54].

Advantages of KeYmaera X implementation approach: the simplicity of LCF-style prover μkernels makes it obvious to see what soundness depends on and what the overall correctness argument is. The resulting prover architecture is modular, thereby separating responsibilities such as the uniform substitution mechanism, axiom indices, unification algorithm, tactics, UI rendering. It is easier to get modularity just right in KeYmaera X. While we believe this to be a general phenomenon in μkernel provers, this success story is slightly modulated by the fact that the third prover design was informed by the prior successes and complications of KeYmaera 3 and KeYmaeraD. The KeYmaera X approach also makes it easier to advance automatic proof principles by adding tactics,

which are, by design, noncritical and open to experimentation and incremental development. Examples of such tactics implemented in KeYmaera X and extended incrementally include ModelPlex [37], component-based verification [40], an axiomatic differential equation solver [50], and ODE automation [59, 66].

Advice for future provers: With the benefit of hindsight, we provide a list of the most impactful decisions and words of advice for future prover designs that can be summarized as justifying that simplicity always wins in the end. The soundness and simplicity of uniform substitution provers cause significant benefits throughout the prover. Immediately invest in common infrastructure such as unification algorithms to simplify tactical implementations down to the essentials and make them robust to change. Invest in proof tree visualization early, because that is an essential debugging aid *during* development. The time spent to develop simple initial proof visualizations is significantly less than the time that their presence saves during prover development. The comparison of KeYmaeraD versus KeYmaera X is an instance of the *bon mot* that premature optimization is the root of all evil, although, admittedly, KeYmaera X still has not achieved anywhere near the distributed proving performance of KeYmaeraD (on the limited continuous dynamics that KeYmaeraD supports). The comparison of all provers clearly indicates that the downstream effects of favoring speed over soundness are self-defeating in the long run. There is a never-ending tension between the desire to build provers in theoretically well-designed elegant programming languages compared to mainstream programming languages. Implementing provers in the former gives more elegant and robust provers, but the frequent lack of libraries requires tedious manual programming effort for data storage, web communication, parsing, IDE integration etc., that are peripheral to core proving.

The most fundamental difference between the three provers comes down to the explicit embrace (KeYmaera 3) or implicit inclusion (KeYmaeraD) of proof rule schemata with side conditions versus the strict adherence to nonschematic axioms or rules without side conditions thanks to uniform substitution (KeYmaera X). Working with schemata makes it easy to add a new reasoning principle but also easy to cause soundness mistakes because a proof rule schema always applies except for an explicit list of exception conditions. This automatic generalization is most easily observed in KeYmaera 3 taclets but also holds for KeYmaeraD's native code implementations of proof rule schemata. Working with nonschematic axioms is better at avoiding soundness glitches by making both syntactic dependencies and generalizations explicit and easier to implement. But working with nonschematic axioms still makes it easy to cause completeness mistakes. Subtle occurrence patterns can make tactics fail to prove provable formulas. Since complicated interaction patterns then manifest as completeness bugs instead of as soundness bugs, they cause less harm and are more easily identified with a failing proof than with a successful proof of an untrue formula.

Acknowledgements. The authors thank Rose Bohrer and the chapter reviewers for helpful feedback on this article.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling **4**(1), 32–54 (2005). doi: 10.1007/s10270-004-0058-x
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016). doi: 10.1007/978-3-319-49812-6
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995). doi: 10.1016/0304-3975(94)00202-T
4. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: Toolympics 2019: An overview of competitions in formal methods. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. pp. 3–24 (2019). doi: 10.1007/978-3-030-17502-3_1, https://doi.org/10.1007/978-3-030-17502-3_1
5. Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Roth, A., Rümmer, P., Schlager, S.: Taclets: A new paradigm for constructing interactive theorem provers. Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM) **98**(1) (2004)
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
7. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In: Furbach, U., Shankar, N. (eds.) IJCAR. LNCS, vol. 4130, pp. 266–280. Springer (2006). doi: 10.1007/11814771_23
8. ter Beek, M., McIver, A., Oliviera, J.N. (eds.): FM 2019: Formal Methods – The Next 30 Years, LNCS, vol. 11800. Springer (2019). doi: 10.1007/978-3-030-30942-8
9. Belta, C., Ivancic, F. (eds.): Hybrid Systems: Computation and Control (part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013. ACM, New York (2013)
10. Bohrer, R., Fernandez, M., Platzer, A.: dL$_\iota$: Definite descriptions in differential dynamic logic. In: Fontaine [16], pp. 94–110. doi: 10.1007/978-3-030-29436-6_6
11. Bohrer, R., Rahli, V., Vukotic, I., Völp, M., Platzer, A.: Formally verified differential dynamic logic. In: Bertot, Y., Vafeiadis, V. (eds.) Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017. pp. 208–221. ACM, New York (2017). doi: 10.1145/3018610.3018616
12. Bohrer, R., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: VeriPhy: Verified controller executables from verified cyber-physical system models. In: Grossman, D. (ed.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018. pp. 617–630. ACM (2018). doi: 10.1145/3192366.3192406
13. Bohrer, R., Tan, Y.K., Mitsch, S., Sogokon, A., Platzer, A.: A formal safety net for waypoint following in ground robots. IEEE Robotics and Automation Letters **4**(3), 2910–2917 (2019). doi: 10.1109/LRA.2019.2923099
14. Church, A.: Introduction to Mathematical Logic. Princeton University Press, Princeton (1956)
15. Doyen, L., Frehse, G., Pappas, G.J., Platzer, A.: Verification of hybrid systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1047–1110. Springer (2018). doi: 10.1007/978-3-319-10575-8_30

16. Fontaine, P. (ed.): International Conference on Automated Deduction, CADE'19, Natal, Brazil, Proceedings, LNCS, vol. 11716. Springer (2019)
17. Frehse, G., Althoff, M. (eds.): ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systemsi, part of CPS-IoT Week 2019, Montreal, QC, Canada, April 15, 2019, EPiC Series in Computing, vol. 61. EasyChair (2019)
18. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 379–395. Springer, Berlin (2011). doi: 10.1007/978-3-642-22110-1_30
19. Fulton, N., Mitsch, S., Bohrer, R., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP. LNCS, vol. 10499, pp. 207–224. Springer (2017). doi: 10.1007/978-3-319-66107-0_14
20. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A., Middeldorp, A. (eds.) CADE. LNCS, vol. 9195, pp. 527–538. Springer, Berlin (2015). doi: 10.1007/978-3-319-21401-6_36
21. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS, Part I. LNCS, vol. 11427, pp. 413–430. Springer (2019). doi: 10.1007/978-3-030-17462-0_28
22. Grebing, S.: User Interaction in Deductive Interactive Program Verification. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2019), https://nbn-resolving.org/urn:nbn:de:101:1-201910300358422776092
23. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Zawadzki, E., Platzer, A.: A formally verified hybrid system for the next-generation airborne collision avoidance system. In: Baier, C., Tinelli, C. (eds.) TACAS. LNCS, vol. 9035, pp. 21–36. Springer (2015). doi: 10.1007/978-3-662-46681-0_2
24. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Schmidt, A., Gardner, R., Mitsch, S., Platzer, A.: A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. STTT **19**(6), 717–741 (2017). doi: 10.1007/s10009-016-0434-1
25. Kouskoulas, Y., Renshaw, D.W., Platzer, A., Kazanzides, P.: Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In: Belta and Ivancic [9], pp. 263–272. doi: 10.1145/2461328.2461369
26. Lange, C., Caminati, M.B., Kerber, M., Mossakowski, T., Rowat, C., Wenzel, M., Windsteiger, W.: A qualitative comparison of the suitability of four theorem provers for basic auction theory. In: Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings. pp. 200–215 (2013). doi: 10.1007/978-3-642-39320-4_13, https://doi.org/10.1007/978-3-642-39320-4_13
27. Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on. IEEE, Los Alamitos (2012)
28. Loos, S.M., Platzer, A., Nistor, L.: Adaptive cruise control: Hybrid, distributed, and now formally verified. In: Butler, M., Schulte, W. (eds.) FM. LNCS, vol. 6664, pp. 42–56. Springer, Berlin (2011). doi: 10.1007/978-3-642-21437-0_6
29. Loos, S.M., Renshaw, D.W., Platzer, A.: Formal verification of distributed aircraft controllers. In: Belta and Ivancic [9], pp. 125–130. doi: 10.1145/2461328.2461350
30. Milner, R.: Logic for computable functions: description of a machine implementation. Tech. rep., Stanford University, Stanford, CA, USA (1972)

31. Mitsch, S., Gario, M., Budnik, C.J., Golm, M., Platzer, A.: Formal verification of train control with air pressure brakes. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) RSSRail. LNCS, vol. 10598, pp. 173–191. Springer (2017). doi: 10.1007/978-3-319-68499-4_12
32. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: Newman, P., Fox, D., Hsu, D. (eds.) Robotics: Science and Systems (2013)
33. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. I. J. Robotics Res. **36**(12), 1312–1340 (2017). doi: 10.1177/0278364917733549
34. Mitsch, S., Passmore, G.O., Platzer, A.: Collaborative verification-driven engineering of hybrid systems. Math. Comput. Sci. **8**(1), 71–97 (2014). doi: 10.1007/s11786-014-0176-y
35. Mitsch, S., Platzer, A.: ModelPlex: Verified runtime validation of verified cyber-physical system models. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV. LNCS, vol. 8734, pp. 199–214. Springer (2014). doi: 10.1007/978-3-319-11164-3_17
36. Mitsch, S., Platzer, A.: The KeYmaera X proof IDE: Concepts on usability in hybrid systems theorem proving. In: Dubois, C., Masci, P., Méry, D. (eds.) 3rd Workshop on Formal Integrated Development Environment. EPTCS, vol. 240, pp. 67–81 (2016). doi: 10.4204/EPTCS.240.5
37. Mitsch, S., Platzer, A.: ModelPlex: Verified runtime validation of verified cyber-physical system models. Form. Methods Syst. Des. **49**(1-2), 33–74 (2016). doi: 10.1007/s10703-016-0241-z, special issue of selected papers from RV'14
38. Mitsch, S., Sogokon, A., Tan, Y.K., Jin, X., Zhan, B., Wang, S., Zhan, N.: ARCH-COMP19 category report: Hybrid systems theorem proving. In: Frehse and Althoff [17], pp. 141–161
39. Mitsch, S., Sogokon, A., Tan, Y.K., Platzer, A., Zhao, H., Jin, X., Wang, S., Zhan, N.: ARCH-COMP18 category report: Hybrid systems theorem proving. In: ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems, ARCH@ADHS 2018, Oxford, UK, July 13, 2018. pp. 110–127 (2018), http://www.easychair.org/publications/paper/tNN2
40. Müller, A., Mitsch, S., Retschitzegger, W., Schwinger, W., Platzer, A.: Tactical contract composition for hybrid system component verification. STTT **20**(6), 615–643 (2018). doi: 10.1007/s10009-018-0502-9, special issue for selected papers from FASE'17
41. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to isabelle/hol. Formal Asp. Comput. **31**(6), 675–698 (2019). doi: 10.1007/s00165-019-00492-1, https://doi.org/10.1007/s00165-019-00492-1
42. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: Olivetti, N. (ed.) TABLEAUX. LNCS, vol. 4548, pp. 216–232. Springer, Berlin (2007). doi: 10.1007/978-3-540-73099-6_17
43. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008). doi: 10.1007/s10817-008-9103-8
44. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1), 309–352 (2010). doi: 10.1093/logcom/exn070
45. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010). doi: 10.1007/978-3-642-14509-4, http://www.springer.com/978-3-642-14508-7
46. Platzer, A.: A complete axiomatization of quantified differential dynamic logic for distributed hybrid systems. Log. Meth. Comput. Sci. **8**(4:17), 1–44 (2012). doi: 10.2168/LMCS-8(4:17)2012, special issue for selected papers from CSL'10

47. Platzer, A.: The complete proof theory of hybrid systems. In: LICS [27], pp. 541–550. doi: 10.1109/LICS.2012.64
48. Platzer, A.: Logics of dynamical systems. In: LICS [27], pp. 13–24. doi: 10.1109/LICS.2012.13
49. Platzer, A.: Differential game logic. ACM Trans. Comput. Log. **17**(1), 1:1–1:51 (2015). doi: 10.1145/2817824
50. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reas. **59**(2), 219–265 (2017). doi: 10.1007/s10817-016-9385-1
51. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer, Cham (2018). doi: 10.1007/978-3-319-63588-0, http://www.springer.com/978-3-319-63587-3
52. Platzer, A.: Uniform substitution for differential game logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR. LNCS, vol. 10900, pp. 211–227. Springer (2018). doi: 10.1007/978-3-319-94205-6_15
53. Platzer, A.: Uniform substitution at one fell swoop. In: Fontaine [16], pp. 425–441. doi: 10.1007/978-3-030-29436-6_25
54. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. Form. Methods Syst. Des. **35**(1), 98–120 (2009). doi: 10.1007/s10703-009-0079-8, special issue for selected papers from CAV'08
55. Platzer, A., Clarke, E.M.: Formal verification of curved flight collision avoidance maneuvers: A case study. In: Cavalcanti, A., Dams, D. (eds.) FM. LNCS, vol. 5850, pp. 547–562. Springer, Berlin (2009). doi: 10.1007/978-3-642-05089-3_35
56. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer, Berlin (2008). doi: 10.1007/978-3-540-71070-7_15
57. Platzer, A., Quesel, J.D.: European Train Control System: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. LNCS, vol. 5885, pp. 246–265. Springer, Berlin (2009). doi: 10.1007/978-3-642-10373-5_13
58. Platzer, A., Quesel, J.D., Rümmer, P.: Real world verification. In: Schmidt, R.A. (ed.) CADE. LNCS, vol. 5663, pp. 485–501. Springer, Berlin (2009). doi: 10.1007/978-3-642-02959-2_35
59. Platzer, A., Tan, Y.K.: Differential equation invariance axiomatization. J. ACM **67**(1), 6:1–6:66 (2020). doi: 10.1145/3380825
60. Quesel, J.D.: Similarity, Logic, and Games - Bridging Modeling Layers of Hybrid Systems. Ph.D. thesis, Department of Computing Science, University of Oldenburg (2013)
61. 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, Berlin (2012). doi: 10.1007/978-3-642-31365-3_34
62. Renshaw, D.W., Loos, S.M., Platzer, A.: Distributed theorem proving for distributed hybrid systems. In: Qin, S., Qiu, Z. (eds.) ICFEM. LNCS, vol. 6991, pp. 356–371. Springer (2011). doi: 10.1007/978-3-642-24559-6_25
63. Rümmer, P., Shah, M.A.: Proving programs incorrect using a sequent calculus for java dynamic logic. In: Gurevich, Y., Meyer, B. (eds.) Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers. LNCS, vol. 4454, pp. 41–60. Springer (2007). doi: 10.1007/978-3-540-73770-4_3, https://doi.org/10.1007/978-3-540-73770-4_3
64. Sogokon, A., Mitsch, S., Tan, Y.K., Cordwell, K., Platzer, A.: Pegasus: A framework for sound continuous invariant generation. In: ter Beek et al. [8], pp. 138–157. doi: 10.1007/978-3-030-30942-8_10

65. Sutcliffe, G., Benzmüller, C., Brown, C.E., Theiss, F.: Progress in the development of automated theorem proving for higher-order logic. In: Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings. pp. 116–130 (2009). doi: 10.1007/978-3-642-02959-2_8, https://doi.org/10.1007/978-3-642-02959-2_8
66. Tan, Y.K., Platzer, A.: An axiomatic approach to liveness for differential equations. In: ter Beek et al. [8], pp. 371–388. doi: 10.1007/978-3-030-30942-8_23
67. Wenzel, M., Wiedijk, F.: A comparison of mizar and isar. J. Autom. Reasoning **29**(3-4), 389–411 (2002). doi: 10.1023/A:1021935419355, https://doi.org/10.1023/A:1021935419355
68. Wiedijk, F.: Comparing mathematical provers. In: Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings. pp. 188–202 (2003). doi: 10.1007/3-540-36469-2_15, https://doi.org/10.1007/3-540-36469-2_15
69. Wiedijk, F. (ed.): The Seventeen Provers of the World, Foreword by Dana S. Scott, LNCS, vol. 3600. Springer (2006). doi: 10.1007/11542384, https://doi.org/10.1007/11542384