KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems*

Nathan Fulton[1], Stefan Mitsch[1], Jan-David Quesel[1],
Marcus Völp[1,2], and André Platzer[1]

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh PA 15213, USA,
{nathanfu,smitsch,jquesel,aplatzer}@cs.cmu.edu,
[2] Technische Universität Dresden, 01157 Dresden, Germany,
marcus.voelp@tu-dresden.de

Abstract. KeYmaera X is a theorem prover for *differential dynamic logic* (dℒ), a logic for specifying and verifying properties of hybrid systems. Reasoning about complicated hybrid systems models requires support for sophisticated proof techniques, efficient computation, and a user interface that crystallizes salient properties of the system. KeYmaera X allows users to specify custom proof search techniques as *tactics*, execute these tactics in parallel, and interface with partial proofs via an extensible user interface.

Advanced proof search features—and user-defined tactics in particular—are difficult to check for soundness. To admit extension and experimentation in proof search without reducing trust in the prover, KeYmaera X is built up from a *small trusted kernel*. The prover kernel contains a list of sound dℒ axioms that are instantiated using a uniform substitution proof rule. Isolating all soundness-critical reasoning to this prover kernel obviates the intractable task of ensuring that each new proof search algorithm is implemented correctly. Preliminary experiments suggest that a single layer of tactics on top of the prover kernel provides a rich language for implementing novel and sophisticated proof search techniques.

1 Introduction

Computational control of physical processes such as cyber-physical systems introduces complex interactions between discrete and continuous dynamics. Developing techniques for reasoning about this interaction is important to prevent software bugs from causing harm in the real world. For this reason, formal verification of safety-critical software is upheld as best practice [4].

Verifying correctness properties about cyber-physical systems requires analyzing the system's discrete and continuous dynamics together in a hybrid system [2]. For example, establishing the correctness of an adaptive cruise control system in a car requires reasoning about the computations of the controller together with the resulting

* This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, NSF CNS-1035800, and CNS-0931985, and by ERC under PIOF-GA-2012-328378 (Mitsch on leave from Johannes Kepler University Linz).
KeYmaera X is available for download from http://keymaeraX.org

physical motion of the car. Theorem proving is a useful technique for proving correctness properties of hybrid systems [11]. Theorem proving complements model checking and reachability analysis, which are successful at finding bugs in discrete systems.

A theorem prover for hybrid systems must be sound to ensure trustworthy proofs, and should be flexible to enable efficient proof search. This paper presents KeYmaera X, a hybrid system theorem prover that meets these conflicting goals. Its design emphasizes a clear separation between a small soundness-critical prover kernel and the rest of the theorem prover. This separation ensures trust in the prover kernel and allows extension of the prover with user-defined proof strategies and custom user interfaces.

We build on experience with KeYmaera [15], an extension of the KeY theorem prover [1]. The success of KeYmaera in cyber-physical systems is due, in part, to its support for reasoning about programs with differential equations and its integration of real arithmetic decision procedures. Case studies include adaptive cruise control and autonomous automobile control, the European Train Control System, aircraft collision avoidance maneuvers, autonomous robots, and surgical robots. Despite the prior successes of KeYmaera, however, its monolithic architecture makes it increasingly difficult to scale to large systems. Aside from soundness concerns, a monolithic architecture precludes extensions necessary for proofs of larger systems, parallel proof search, or proof strategies for specific analyses such as model refinement or monitor synthesis.

KeYmaera X is a clean-slate reimplementation to replace KeYmaera. KeYmaera X focuses on a small trusted prover kernel, extensive tactic support for steering proof search, and a user interface intended to support a mixture of interactive and automatic theorem proving. KeYmaera X improves on automation when compared to KeYmaera for our ModelPlex case study: it automates the otherwise $\approx$60% manual steps in [8].

2 KeYmaera X Feature Overview

Hybrid Systems. Hybrid dynamical systems [2,12] are mathematical models for analyzing the interaction between discrete and continuous dynamics.

Hybrid automata [2] are a machine model of hybrid systems. A hybrid automaton is a finite automaton over an alphabet of real variables. Variables may instantaneously take on new values upon state transitions. Unlike classical finite automata, each state is associated with a continuous dynamical system (modeled using ordinary differential equations) defined over an evolution domain. Whenever the system enters a new state, the variables of the system evolve according to the continuous dynamics and within the evolution domain associated with that state. Hybrid automata are not conducive to *compositional* reasoning; to establish a property about a hybrid automaton, it does not suffice to establish that property about each component of a decomposed system.

Hybrid programs [10,11,12], in contrast, are a compositional programming language model of hybrid dynamics. They extend regular programs with differential equations. A syntax and informal semantics of hybrid programs is given in Table 1.

Differential Dynamic Logic. Differential dynamic logic ($\mathsf{d}\mathcal{L}$) [10,11,12] is a first-order multimodal logic for specifying and proving properties of hybrid programs. Each hybrid program α is associated with modal operators $[\alpha]$ and $\langle\alpha\rangle$, which express state reachability properties of the parametrizing program. For example, $[\alpha]\phi$ states that the formula

Table 1. Hybrid Programs

Program Statement	Meaning
$\alpha;\beta$	Sequential composition of α and β.
$\alpha \cup \beta$	Nondeterministic choice ($\cup$) executes either α or β.
α^*	Repeats α zero or more times.
$x := \theta$	Evaluate the expression θ and assign its result to x.
$x := *$	Assigns some arbitrary real value to x.
$\{x_1' = \theta_1, ..., x_n' = \theta_n \& F\}$	Continuous evolution along the differential equation system $x_i' = \theta_i$ for an arbitrary duration within the region described by formula F.
$?F$	Tests if formula F is true at current state, aborts otherwise.

ϕ is true in any state reachable by the hybrid program α. Similarly, $\langle\alpha\rangle\phi$ expresses that the property ϕ is true after some execution of α. The dL formulas are generated by the EBNF grammar

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

where θ_i are arithmetic expressions over the reals, ϕ and ψ are formulas, α ranges over hybrid programs, and $\sim$ is a comparison operator $=, \neq, \geq, >, \leq, <$.

Example 1. The following dL formula describes a safety property for a car model.

$$\underbrace{v \geq 0 \wedge A > 0}_{\text{initial condition}} \rightarrow [\big(\underbrace{(a := A \cup a := 0)}_{\textit{ctrl}} ; \underbrace{\{p' = v,\ v' = a\}}_{\textit{plant}}\big)^*] \underbrace{v \geq 0}_{\text{postcondition}} \tag{1}$$

Formula (1) expresses that a car, when started with non-negative velocity $v \geq 0$ and positive acceleration $A > 0$ (left-hand side of the implication), will always drive forward ($v \geq 0$) after executing $(\textit{ctrl};\textit{plant})^*$, i.e. running *ctrl* followed by the differential equation *plant* arbitrarily often. Since there are no evolution domain constraints in $plant$ that limit the duration, each continuous evolution has an arbitrary duration $r \in \mathbf{R}_{\geq 0}$. As its decisions, *ctrl* lists that the car can either accelerate $a := A$ or coast $a := 0$, while *plant* describes the motion of the car (position p changes according to velocity v, velocity v according to the chosen acceleration a). Details on dL are in the literature [10,11,12], including a tutorial on modeling and proving in KeYmaera [16].

Proofs in KeYmaera X. Proofs in KeYmaera X are built up from three components (kernel primitives): a small set of dL axioms (not axiom schemata) [14] from its axiomatization [12], bound variable renaming and uniform substitution [13,14], and the propositional fragment of the dL sequent calculus [10]. Even if unnecessary in theory [12,14], the propositional fragment of the dL sequent calculus is included in the prover kernel because the implementation is easy to check for soundness and significantly improves the efficiency of the prover during proof search.

The KeYmaera X prover kernel implements a Hilbert system for dL [12] as a uniform substitution calculus with bound variable renaming and uniform substitution [14]. A typical proof in KeYmaera X involves a succession of cuts of axioms, followed by uniform substitution and variable renaming to align the current goal with the cut-in axiom, and use the instantiated axiom by fast contextual equivalence rewriting [14].

Table 2. Dynamics of tactic combinators

Tactic Combinator	Meaning
$t ::= b$	b Basic tactics.
$\mid\ t \ \& \ u$	Executes t and, if successful, then executes u.
$\mid\ t \mid u$	Executes t only if t is applicable. If t is not applicable, then u is executed.
$\mid\ t^*$	Repeats t until t is no longer applicable.
$\mid\ <(u_1, \ldots, u_k)$	Applied to a goal with k subgoals, each u_i is executed on the i^{th} subgoal.
$\mid\ \texttt{label}(\ell)$	Labels the current goal with label ℓ.
$\mid\ \texttt{onLabel}(\ell, t)$	Executes tactic t only if the goal is labeled ℓ.
$\mid\ \texttt{ifT}(c)(u, v)$	Executes u if c is true, and executes v otherwise.

Kernel Primitives and the dL Sequent Calculus. Although the Hilbert-style prover kernel is helpful for ensuring soundness, manually constructing proofs from kernel primitives is prohibitively tedious. To automate proof construction, KeYmaera X provides a library of basic tactics and a set of tactic combinators.

Basic tactics implement the dL sequent calculus [10,11] in terms of kernel primitives. Some dL proof rules are trivial to implement in terms of kernel primitives; for example, `ImplyRight` is a tactic that just applies the corresponding proof rule in the kernel's propositional sequent calculus implementation. Other dL sequent rules compose multiple prover kernel primitives (e.g., the Differential Invariant proof rule [14] for proving properties of differential equations without solving them).

Tactical Proving. The tactic combinator language (see Table 2) provides a mechanism for combining basic and other pre-existing tactics to build proof search strategies. All tactics—whether built-in or constructed using combinators—are applied to a sequent or a set of sequents called a goal. Tactics have an applicability condition and a dynamic semantics, both of which may depend upon the goal to which the tactic is applied.

The applicability condition associated with each tactic defines a set of sequents at which the tactic may *possibly* succeed. Applicability for built-in tactics is defined by their author, and these applicability conditions extend automatically to terms of the combinator language. The dynamic semantics of a tactic is ultimately a sequence of kernel primitives that are applied to the current goal. All tactics may either succeed or fail on error, and errors are propagated through combinator terms.

The sequential composition combinator ($t \& u$) is similar to the semi-colon in a C-like programming language, and is used in a similar way. The tactic $t \& u$ is applicable when the first tactic (t) is applicable. The tactic results in an error under three conditions: if t results in an error, if u is not applicable at the result of t, or if u results in an error.

The either combinator ($t|u$) is useful when writing tactics that apply at many possible syntactic forms (e.g., a tactic that symbolically executes any hybrid program). It is applicable when either t or u is applicable. The applicable tactic is executed and the other is ignored; if both are applicable, then t is executed and u is ignored. The tactic $t|u$ results in an error if the executed tactic results in an error.

The Kleene star (t^*) saturates the tactic t by applying t as often as possible, which is useful when writing general-purpose tactics. The tactic t^* is always applicable and results in an error if any iteration of t results in an error.

Branching composition ($<(u_1, ..., u_k)$) is useful for handling branching proofs (e.g., any proof that uses invariants or involves disjunctive assumptions). The tactic is always applicable, and errors when applied to a goal with a non-k number of subgoals or if any u_i is inapplicable or results in an error. Branching ($<(u_1, ..., u_k)$) has a sequential semantics given by applying each u_i sequentially. The parallel semantics of branching depends upon scheduling and synchronization, which are defined in terms of a proof tree with And/Or-branching as in Fig. 1. KeYmaera X's proof search engine is discussed in Sect. 3.

Finally, labels are useful for structuring branching proofs. Many built-in tactics that generate multiple subgoals provide labels for each subgoal, which can be matched against using the `onLabel` combinator. The tactic `onLabel`$((\ell_1, t_1), ..., (\ell_k, t_k))$ is applicable if any of the labels ℓ_i exists in the current goal and executes the corresponding constituent tactic t_i, resulting in an error if t_i results in an error.

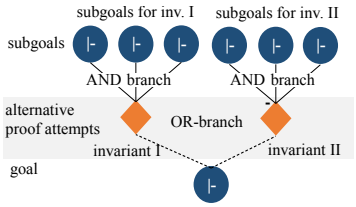

Fig. 1. Proof tree data structure

Proof search strategies are expressed using combinators. While generic proof search strategies exist (e.g., `Master`), KeYmaera X allows user-defined custom proof search strategies expressed as tactics. The full Scala language is available when implementing proof search strategies, but KeYmaera X also exposes an interface for running pure combinator tactics. Where automated tactics fail, users can interact with the prover by manually applying proof rules or by selecting the appropriate tactic and any necessary input (e.g., loop invariants). The following tactic example illustrates the tactic language by providing a detailed strategy for proving the safety property of Example 1 (note, that the tactic `Master` with invariant $v \geq 0$ would prove the example fully automatically as well but it is instructive to see the shape of the proof in a detailed proof tactic).

```
ImplyRight & Loop("v>=0") & onLabel(
  ("base␣case", Master),
  ("induction␣step", ImplyRight & Seq & Choice & AndRight & <
    (Assign & ODESolve & Master,
     Assign & ODESolve & Master) ),
  ("use␣case", Master) )
```

At every execution step the strategy applies to the topmost operator, starting with the implication in (1) followed by induction with invariant $v \geq 0$ to handle the loop in the box modality. The loop induction tactic generates three labeled subgoals.

The subgoals labeled "base case" and "use case" are handled by the `Master` tactic, a general-purpose tactic for proving dL formulas. `Master` tries non-branching propositional tactics and hybrid program tactics, then applies any branching in propositional tactics, then searches for invariants, and finally resorts to quantifier elimination.

The tactic for the induction step follows the structure of the program. `Seq` handles the sequential composition between *ctrl* and *plant*, then `Choice & AndRight` split the non-deterministic choice $a := A \cup a := 0$. On the resulting two sub-branches, the assignments $a := A$ and $a := 0$ are handled, followed by `ODESolve`, which solves the differential equations of *plant*. The remaining nonmodal goals are proved by `Master`.

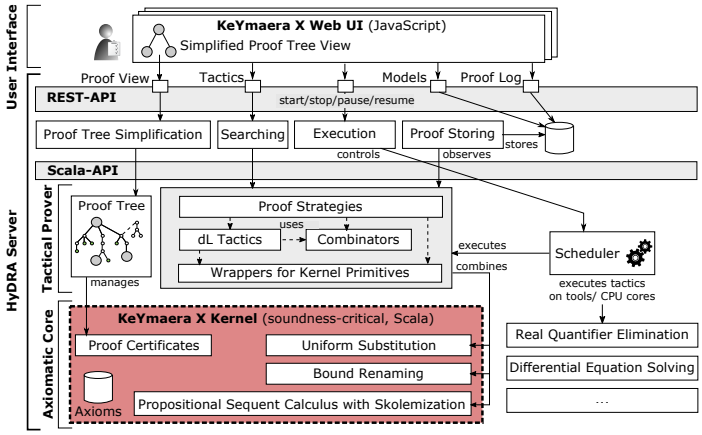

Fig. 2. KeYmaera X architecture: soundness-critical kernel is shown in dark with a dashed border

3 KeYmaera X Tool Architecture

KeYmaera X was designed to achieve powerful automation of hybrid systems theorem proving while ensuring soundness. The architecture of KeYmaera X (Figure 2) is separated into a small, soundness-critical kernel and an extensive tactic framework to regain and exceed the convenience of powerful proof rules. A scheduler multiplexes tactics to worker threads to utilize available CPU cores. It also manages calls to external tools, such as real quantifier elimination and differential equation solving. On top of proof tactics and scheduling, the HyDRA server provides components for proof tree simplification, tactic search and custom tactic scheduling policies, as well as for storing and accessing proofs. These components can be accessed remotely through a REST-API. The KeYmaera X web user interface, implemented in JavaScript, uses this REST-API to communicate with the server. The remaining subsections are organized around Fig. 2.

HyDRA: Hybrid Distributed Reasoning Architecture. KeYmaera X has an isolated prover kernel, which offers a restricted interface to the remaining system components. The prover kernel operates in terms of *proof certificates*, which capture certified provability in the kernel. A proof certificate means that from certain premises the prover can soundly show a particular conclusion (e.g., a rule `AndRight` would have two premises, one for each conjunct, whereas an axiom has no premises). KeYmaera X ensures soundness by construction; it disallows construction of proof certificates that do not correspond to a correct derivation. That way, the prover kernel does not need to care about how proof certificates relate to each other, as long as it ensures that proof certificates only originate from within the kernel. To achieve this, components outside the soundness-critical kernel, such as tactics, the user interface and the framework for parallel execution, receive at most read-only access to proof certificates. All mechanisms for creating new proof certificates—rewrites corresponding to the axioms of $\mathsf{d}\mathcal{L}$, uniform substitution, bound variable renaming, Skolemization and the rules of the propositional sequent calculus—are contained in the kernel. Proof certificates are managed

in an And/Or proof tree outside the prover kernel, so that tactics and users have access to the proof history (Fig. 1 denotes And-branches with solid lines between nodes in the proof tree, whereas Or-branches are depicted using dashed lines).

Correctness of the prover depends on the soundness of Scala's pattern matching capabilities in a similar way that Isabelle [9] depends upon the correctness of Standard ML. Our selection of Scala is motivated by our need to interact with Mathematica and a web server. The Scala ecosystem is also attractive from the perspective of supporting parallel proof search and other advanced proof search features.

Collaboration and Distributed Search. KeYmaera X supports collaborative proving and parallel, distributed proof search through a client-server architecture and proof tree data structures with Or-branching. Multiple user interfaces may interact with the prover via a REST-API on different goals, or attempt different strategies on the same goal.

Similarly, multiple goals may be processed in parallel and multiple tactics tried on the same goal. KeYmaera X supports parallel exploration of proof strategies by means of Or-branching alternatives in the proof-tree data structure and by its continuation-passing tactics library, which we explain in greater detail below.

KeYmaera X Kernel. The soundness-critical KeYmaera X kernel consists of: (i) algebraic data types representing $\mathsf{d}\mathcal{L}$ expressions and proof certificates; (ii) the axioms of differential dynamic logic [14]; (iii) bound variable renaming and uniform substitution rules [14]; (iv) a propositional sequent calculus with Skolemization [10]. To a lesser extent, the kernel also features expression parsing and printing. KeYmaera X bans them from the soundness-critical kernel by dynamically checking whether pretty-printing reparses to the original expressions and by declaring the pretty-printed property to be proved rather than the textual representation in input files.

The entire prover kernel has a size of about 1700 lines of Scala code (LOC). Parsing and printing weighs in at another 1700 LOC. Proofs are certified by an LCF-style design in which only the small list of certified proof rules can create proof certificates. All this puts verifying the kernel in feasible range: The axiomatic portion of the kernel uses primarily algebraic data types and recursive functions defined over these types, so mechanizing the theory of KeYmaera X in a higher-order proof assistant and possibly performing code extraction appears feasible.

KeYmaera X implements rules from the propositional sequent calculus, bound variable renaming, and, most importantly, uniform substitution. These rules are the basis for constructing all proofs. Tactics are constructed from axioms by aligning them with the current goal using bound variable renaming and uniform substitution. The axiom base from which proofs are constructed is kept small (49 axioms and 17 additional derived axioms) and syntactically close to the way it is presented in papers and books. Since the axioms cannot be proven within the system itself, this design is crucial to allow manual inspection to ensure that the system's foundation is sound and well chosen.

KeYmaera X relies on external tools as real arithmetic decision procedures. Arithmetic facts are stored as lemmas that are verified by the decision procedures. These lemmas are collected together with the resulting proof and, thus, can be fed into different decision procedures to increase trust in their correctness or retained as arithmetic assumptions. The dependence on external tools is minimized compared to KeYmaera [15].

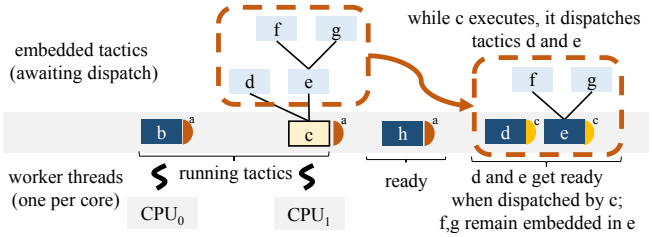

Fig. 3. Tactic scheduling using continuations

Differential equation solvers are removed from the trusted kernel and arithmetic is used exclusively at the leaves of the proof tree.

Runtime and Scheduler for Executing Tactics. Tactics and kernel primitives (through their wrappers) as well as external tools are not invoked directly from the user interface but passed to a scheduler. The scheduler multiplexes tactics to worker threads for parallel execution and manages limited parallelism and blocking on external tools.

To achieve this, the scheduler instantiates one worker thread per CPU core and in addition one worker thread for each blocking link to external tools. By blocking we mean a link that requires the worker thread to wait for a result after it has passed the request to a tool. In addition, KeYmaera X tactics are schedulable objects comprised of a main body and a continuation, which can be passed to other tactics to regain control after completion, in particular if they have been executed on a different CPU core.

Figure 3 illustrates the dispatching of tactics and the role of continuations. A tactic a (not shown) has dispatched the tactics b, c and h for parallel execution by inserting them into the global priority-sorted ready list from which the worker threads on CPU_1 selected c, which it currently executes. Worker threads always pick the highest-prioritized ready tactic from the ready list and execute them non-preemptively (i.e., they first complete a started tactic before they look for the next one). Tactic c represents any tactic that would add multiple independent tactics to the queue, such as the $<(d, e)$ tactic. The tactics d and e are associated with subgoals of the goal at which c is applied. Once a tactic has been associated to a proof node and a continuation, the tactic is ready for dispatch into the scheduler's ready list. The result of dispatching of d and e is shown on the right of Fig. 3 when following the arrow. Tactic e is a combinator (e.g., $e = f \,\&\, g$) with embedded tactics f, g. Because e did not yet execute and because g will execute on the subgoal yet-to-be produced by f, these tactics are not ready yet.

To regain control after d and e complete, c has passed a continuation to both tactics (c is the parent of the continuation). Continuations are invoked once the body of a tactic completes. A continuation can inspect the result and the completion status (success or failure) of the completed tactic, as well as its parent to make decisions about the next proof step based on whether or not the proof changed.

User Interface. The KeYmaera X system features multiple interfaces: (i) a Scala-API for accessing the axiomatic core and tactical prover programmatically from (standalone) Scala and Java applications; (ii) a REST-API intended for remote access to the HyDRA

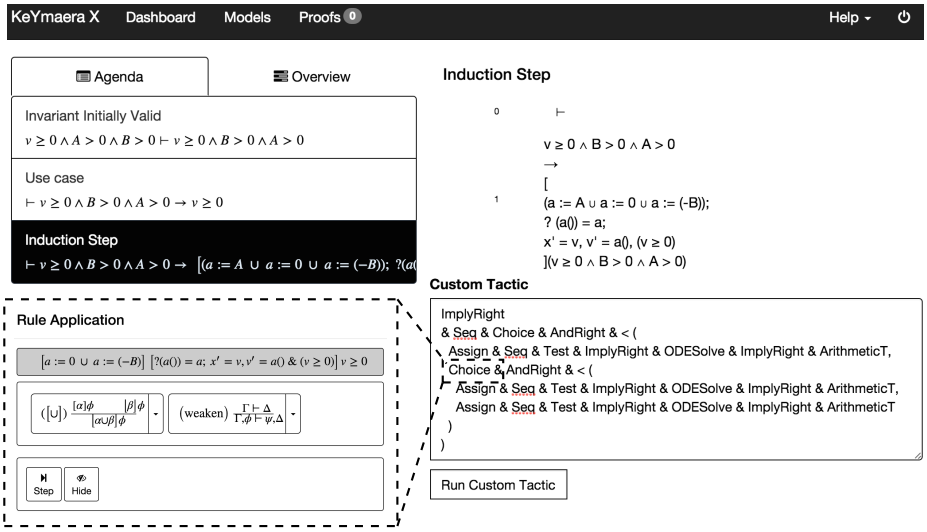

Fig. 4. A tactic for closing the induction step of a simple hybrid car model. The dotted selection illustrates what the Apply Rule dialog would look like just before executing the second `Choice` in the custom tactic.

server; and (iii) a graphical web-based user interface for point-and-click interaction. The Scala-API is designed for tight integration of KeYmaera X into other programs. It is the basis for the HyDRA server and used in the development process for unit testing. The REST-API wraps the Scala-API in a web application and gives access to server functionality: it identifies the "resources" at the HyDRA server (such as goals in a proof tree, formulas in a sequent, and tactics) using hierarchical URLs and uses standard HTTP requests to manipulate these resources. On top of that, KeYmaera X provides a native web interface for managing proofs and lemma databases, as well as for interactive and tactical proving sessions. Figure 4 shows the web interface during an interactive proving session. In the web interface, proof trees are collapsed for presentation into simplified views, which highlight proof steps at the granularity of dℒ sequent rules but shortcut through the axiom-application steps that we introduced to improve confidence in soundness. Custom tactics can be specified using the combinator language of Sect. 2. Alternatively, proof rules such as ODESolve can be selected directly by clicking on the formula, as illustrated in Fig. 4.

4 Related Work

KeYmaera X is the first theorem prover to unify Hilbert systems and Gentzen-style sequent calculi by combining uniform substitution with a flexible tactics mechanism. Hilbert systems simplify reasoning about soundness, which reduces the complexity and risk associated with extending the theorem prover with new proof search techniques or new logic fragments. This distinction separates KeYmaera X from other deductive verification systems such as KeY [3] and KeYmaera [15].

LCF-style theorem provers, including Isabelle [9], feature both a minimal trusted kernel as well as support for tactics. These tools influenced the design of KeYmaera X. Most major theorem provers, including Coq [7] and Isabelle [9], also provide user interfaces. In [5], similar to KIV [6], a tactical theorem prover for verifying software is presented. Unlike these, KeYmaera X is particularly well-suited to the analysis of hybrid dynamical systems with their differential equations.

Other successful tools exist for hybrid systems; however, apart from KeYmaera, none based on the rigor of a sound logic let alone a small kernel. A comparison of $\mathsf{d}\mathcal{L}$ with other approaches to analysis of hybrid systems is provided in the literature [11].

Acknowledgments. The authors thank the anonymous reviewers for their helpful feedback, and Ran Ji for help with testing and extending KeYmaera X.

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)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems. pp. 209–229 (1992)
3. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of Object-oriented Software: The KeY Approach. Springer-Verlag, Berlin, Heidelberg (2007)
4. Bowen, J., Stavridou, V.: Safety-critical systems, formal methods and standards. Software Engineering Journal 8(4), 189–209 (Jul 1993)
5. Felty, A.P., Howe, D.J.: Tactic theorem proving with refinement-tree proofs and metavariables. In: Bundy, A. (ed.) CADE. LNCS, vol. 814, pp. 605–619. Springer (1994)
6. Heisel, M., Reif, W., Stephan, W.: Tactical theorem proving in program verification. In: Stickel, M.E. (ed.) CADE. LNCS, vol. 449, pp. 117–131. Springer (1990)
7. The Coq development team, organization = LogiCal Project: The Coq proof assistant reference manual (2004), http://coq.inria.fr, version 8.0
8. 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)
9. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
10. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. 41(2), 143–189 (2008)
11. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
12. Platzer, A.: Logics of dynamical systems. In: LICS. pp. 13–24. IEEE (2012)
13. Platzer, A.: Differential game logic. CoRR abs/1408.1980 (2014)
14. Platzer, A.: A uniform substitution calculus for differential dynamic logic. In: Felty, A., Middeldorp, A. (eds.) CADE. LNCS, Springer (2015)
15. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer (2008)
16. Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A.: How to model and prove hybrid systems with KeYmaera: A tutorial on safety. STTT (2015)