KeYmaera X Tutorial

André Platzer

Computer Science Department
Carnegie Mellon University
Pittsburgh, USA

This document is updated less frequently than its web version

https://keymaeraX.org/Xtutorial.html

Contents

1 Differential Dynamic Logic in KeYmaera X **5**
1.1 Terms 6
1.2 Logical Connectives 7
1.3 Modalities 9
1.4 Operator Precedence for Formulas 10
1.5 Optional: Function Symbols 11
1.6 Optional: Predicate Symbols 14

2 Hybrid Programs in KeYmaera X **17**
2.1 Hybrid Program Statements 18
2.2 dL Modalities with Hybrid Programs 21
2.3 Operator Precedence for Programs 21
2.4 Optional: Nondeterministic Assignments 22
2.5 Optional: Program Symbols for Subprograms 23

3 Proofs in KeYmaera X **25**
3.1 Propositional Proofs 26
3.2 Quantifier Proofs 29
3.3 Dynamics Proofs 31
3.4 Loop Invariant Proofs 32
3.5 @Invariant Annotation 33
3.6 Optional: Chase-based Proofs 34
3.7 Optional: Reasoning in Context 35
3.8 Optional: Monotone Generalization 37
3.9 Advanced: Loop Variant Proofs 39
3.10 Advanced: Loop Fixpoint Proofs 42

4 Differential Equation Proofs in KeYmaera X **45**
4.1 Solving Differential Equations 46
4.2 Invariants for Differential Equations 47
4.3 Differential Invariants 48
4.4 Differential Cuts 51
4.5 @Invariant Annotation 54
4.6 Differential Weakening 54
4.7 Conserved Quantities 56
4.8 Differential Ghosts 58

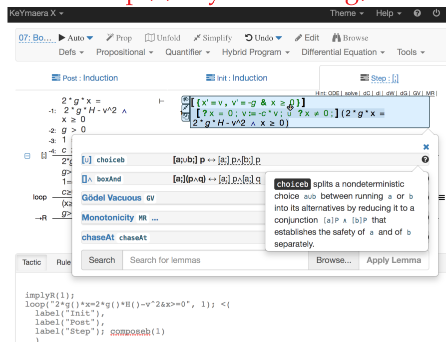
Chapter 1

Differential Dynamic Logic in KeYmaera X

Welcome to the KeYmaera X Tutorial in which you will learn how to use the KeYmaera X aXiomatic Tactical Theorem Prover for Hybrid Systems from a pragmatic perspective.

KeYmaera X KeYmaera X is a theorem prover for *differential dynamic logic* (dL), a logic for specifying and verifying properties of *hybrid systems* with mixed discrete and continuous dynamics. This tutorial provides practical tool aspects and is complementary to the textbook *Logical Foundations of Cyber-Physical Systems*, in which provides comprehensive information on differential dynamic logic can be found. KeYmaera X is available at

http://keymaeraX.org/



Part Summary This part will give you an opportunity to explore how to use the simpler logical operators of differential dynamic logic in KeYmaera X. We will first focus only on propositional connectives such as conjunction and implication as well as quantifiers of first-order logic before proceeding to actual hybrid systems in the next chapter. This preparation gives you a chance to first familiarize yourself with the handling of usual first-order logic. The defining modalities of differential dynamic logic will only play a subordinate role in this chapter, before flourishing in later parts of this tutorial.

Background This tutorial assumes that you have read or refer to the following chapters in the *Logical Foundations of Cyber-Physical Systems* textbook for background information on the principles as needed:

- Chapter 4: Safety & Contracts
- Chapter 2: Differential Equations & Domains Sections 2.6+2.7

1.1 Terms

Real arithmetic terms. KeYmaera X provides standard operators of real arithmetic. In theory, dL terms are polynomial terms but many others are definable. Thus, KeYmaera X also allows division as long as you pay attention not to divide by zero.

Term Operators of Differential Dynamic Logic (dL).

dL	KeYmaera X	Operator	Meaning
x	`x`	variable	value of variable x in the current state
0.5	`0.5`	constant	value of the numeric constant, here, 0.5 etc.
$-e$	`-e`	negative	negative of the value of terms e
$e+d$	`e+d`	plus	sum of values of terms e and d
$e-d$	`e-d`	minus	difference of values of terms e and d
$e \cdot d$	`e*d`	times	product of values of terms e and d
e/d	`e/d`	divide	division of values of terms e and d (make sure $d \neq 0$)
e^n	`e^n`	power	value of term e raised to the power of n
$f(e1, ..., ek)$	`f(e1,...,ek)`	function	value of f at values of $(e1, ..., ek)$
$(e)'$	`(e)'`	differential	value of differential of e

Operator precedences and associativity are as usual in mathematics.

Operator precedence. Multiplicative operators $(\cdot, /)$ bind stronger than additive operators $(+, -)$ and arithmetic operators are left-associative except for powers. Hence, parentheses can be left out as usual, e.g., $a + b \cdot c$ is $a + (b \cdot c)$ and $a - b - c$ is $(a - b) - c$. The primary difference of the mathematical notation in dL compared to the ASCII rendition in KeYmaera X are the keyboard characters for multiplication and the inline notation for the power operator.

Variables versus functions. Variables such as x can change their value while a cyber-physical system is evolving. That makes sense since the position x of your car may also be different today than it is tomorrow. Function symbols f, even the ones with 0 arguments, instead do not change their value. So the value of $f(e)$ depends on the value of e. But if e and d have the same value then so will $f(e)$ and $f(d)$. For function symbols f of 0 arguments that, in particular, means that, unlike a variable x, the term $f()$ has the same value throughout the evolution of your system (1.5). During differential equations, differential variables x' with primes denote the time-derivative of x, but outside differential equations, you can just think of x' is just another variable.

Example 1.1 (Mass-energy equivalence)**.** If variable E denotes energy, m denotes mass and c is the speed of light, then Einstein's famous equivalence of energy and mass can be

expressed by this term being zero:

$$E - mc^2$$

The corresponding term in KeYmaera X is transliterated into ASCII syntax with an explicit multiplication operator * and exponentiation operator ^ written inline:

E−m*c^2

Example 1.2 (Divisions)**.** The midpoint between a and b can be computed as the term $(a + b)/2$ so the sum of a and b divided by 2. Only whenever you divide you had better take care that the division was meaningful. This division is perfectly harmless, because it is a division by 2. But the division $(a + b)/x$ is only meaningful in a context where you have made sure that x cannot be 0. Similar considerations come up if you try to use rational powers such as $x^{0.5}$, which is the same as $\sqrt{x}$, thus is only meaningful for nonnegative x. It is generally best to transform questions to stay in polynomial arithmetic or make sure that you guard expressions suitable and that such transformations can be done automatically.

1.2 Logical Connectives

Classical logic operators. KeYmaera X provides standard operators of classical propositional logic and the quantifiers of first-order logic of real arithmetic. The only difference between practice in KeYmaera X and principles in dL is the ASCII notation that KeYmaera X uses. A succinct summary of the syntax and semantics is on the KeYmaera X Cheat Sheet.

Operators of Differential Dynamic Logic (dL).

dL	KeYmaera X	Operator	Meaning
$e = d$	`e=d`	equals	values of terms e and d are equal
$e \geq d$	`e>=d`	greater-or-equal	value of e greater-or-equal to value of d
$p(e1, ..., ek)$	`p(e1,...,ek)`	predicate	p holds for the value of $(e1, ..., ek)$
$\neg P$	`!P`	not	P false
$P \land Q$	`P & Q`	and conjunction	both P and Q are true
$P \lor Q$	`P | Q`	or disjunction	P true or if Q true
$P \to Q$	`P -> Q`	implies	P false or Q true
$P \leftrightarrow Q$	`P <-> Q`	equivalent	P and Q both true or both false
$\forall x\, P$	`\forall x P`	all quantifier	P true for all real values of variable x
$\exists x\, P$	`\exists x P`	exists quantifier	P true for some real value of variable x
$[a]P$	`[a]P`	box $[\cdot]$	P true after all runs of HP a
$\langle a\rangle P$	`<a>P`	diamond $\langle\cdot\rangle$	P true after at least one run of HP a

Unary operators (including $\forall x$, $\exists x$, $[a]$, $\langle a\rangle$) bind stronger than binary operators.

Operator precedence. Unary operations (including $\neg$, quantifiers $\forall x$, $\exists x$, modalities $[a]$,$\langle a\rangle$) bind more strongly than binary operators. We let $\land$ bind more strongly than $\lor$, which binds more strongly than $\to$, $\leftrightarrow$. All logical operators associate to the right. The primary difference of the mathematical notation in dL compared to the ASCII rendition in KeYmaera X are the keyboard characters for the logical connectives and the spelled-out LaTeX style quantifiers. When you load dL formulas in KeYmaera X it will display it as in dL, so for example quantifier `\forall` turns into $\forall$ in a proof.

Note (Modalities). The defining formula operators of differential dynamic logic, the modalities $[a]$ and $\langle a \rangle$, are crucial. But their treatment will be deferred till the next tutorial part (2), where it will become clear how hybrid systems a are described. For now it suffices to read $[a]P$ as "always after running hybrid system a from the current state is formula P true" without worrying too much about a. Likewise, $\langle a \rangle P$ can be read as "there is a way of running hybrid system a from the current state after which formula P is true."

Example 1.3 (Propositional arithmetic)**.** Discarding modalities for the time being and returning to the propositional logical connectives consider a simple formula using propositional logic and arithmetic:

$$v^2 > 0 \to \big(x \neq 0 \leftrightarrow (xv > 0 \lor xv < 0)\big)$$

This formula expresses that if the square of v is positive, then x is nonzero if and only if the product of xv is positive or is negative. The corresponding formula in KeYmaera X is transliterated into ASCII syntax with an explicit multiplication operator `*`:

```
v^2>0 -> (x!=0 <-> (x*v>0 | x*v<0))
```

Load this example in KeYmaera X and by pasting it into a **KeYmaera X->New Model** then enter a name. Click **Start Proof** and prove it automatically by clicking **Auto**. If you have configured KeYmaera X correctly it should prove immediately.

Example 1.4 (Counterexamples)**.** Now try what happens when you drop the assumption $v^2 > 0$:

$$x \neq 0 \leftrightarrow (xv > 0 \lor xv < 0)$$

The corresponding formula in KeYmaera X is transliterated into ASCII syntax:

```
x!=0 <-> (x*v>0 | x*v<0)
```

Load this example in KeYmaera X and click **Start Proof** and **Auto**. This time, KeYmaera X took some proof steps and reduced the problem to the impossible task of showing `|- false` and pointed you to the fact that it found a counterexample by marking the proof branch with a flash. To search for counterexamples click **Tools->Counterexample** and inspect the counterexample, e.g., $v = 0, x = -1$ which falsifies the above formula. Indeed, the assumption $v^2 > 0$ of Example 1.3 was necessary to rule out this counterexample.

Don't get confused when your proof reduces to the question of proving `|- false`. You cannot prove the contradiction `false`, but, sure, if only you could, then your original question would be proved as well. Depending on what proof steps were used, the premise `|- false` either means that your original question is also not provable (if only equivalence transformations were used) or it merely means that you may have used the wrong proof attempt and should try something else instead (if implicational transformations were used in the proof).

Example 1.5 (Quantified arithmetic)**.** More interesting logical formulas involve quantifiers of first-order logic. Recall that quantifiers always quantify over all real numbers in differential dynamic logic, because makes the most sense in hybrid systems or cyber-physical systems applications. Consider the following first-order logic formula with quantifiers and arithmetic:

$$v^2 > 0 \to \forall x \, \big(x \neq 0 \to ((\exists c \, xvc^2 = 1) \lor (\exists c \, xvc^2 = -1))\big)$$

This formula expresses that if the square of v is positive, then for all real numbers x if x is nonzero then there is a real number c such that the product of x and v and the square of c is 1 or there is a real number c such that the product of x and v and the square of c is -1. It may have been clearer to use two different existentially quantified variables c and d in the two disjuncts but it is perfectly allowed to reuse variable names in multiple places. As usual, it is always the inner-most scope that a variable refers to. The corresponding formula in KeYmaera X is transliterated into ASCII syntax with an explicit multiplication operator `*` and the quantifiers written as `\forall` or `\exists`, respectively:

```
v^2>0 -> \forall x (x!=0 -> ((\exists c x*v*c^2=1) | (\exists c x*v*c^2=-1)))
```

Load this example in KeYmaera X and click **Start Proof** and prove it automatically by clicking **Auto**. If you have configured KeYmaera X correctly it should prove immediately.

1.3 Modalities

dL Modalities. Playing around with these first-order logic formulas is fun, but things will get a lot more interesting once the genuine power of differential dynamic logic will be exploited in the next chapter: the *modalities* $[\alpha]$ and $\langle\alpha\rangle$. The box modality $[\alpha]$ can be applied to any dL formula P, giving a new dL formula $[\alpha]P$, which expresses that formula P is true after all runs of α. Likewise, the diamond modality $\langle\alpha\rangle$ can be applied to any dL formula P, giving a new dL formula $\langle\alpha\rangle P$, which expresses that P is true after some run of α (at least one run).

For example, when γ is the hybrid systems model of your car, the dL formula $[\gamma]\, v > 0$ expresses that *all* behavior of the hybrid system γ is such that the velocity v is positive. Don't get carried away, though, no matter how speedy such a car controller may sound, you probably want to modify the controller in γ to make sure this property is *not* true! In order to avert a crash with another car, your controller had best hit the brakes and stop at velocity 0. Distinguishing these will be the challenge for upcoming chapters, though.

Programs postponed. The next logical step is to explore the role of the modalities in differential dynamic logic that make it possible to talk about the properties of all ($[a]P$) or about some ($\langle a\rangle P$) behaviors of hybrid system a. If you have read Chapter 3: Choices & Control, then you already know that a can be any hybrid program. The exact description of the hybrid system a will be the topic of the next chapter, however.

Example 1.6 (Abstract box modalities)**.** For now, simply suppose that c is a hybrid systems model for your car. Then the following dL formula expresses that all runs of your car model c have a positive velocity:

$$[c]\, v > 0$$

That is, this formula expresses: after all ways of running the hybrid system c is the formula $v > 0$ true. In fact, when you think about it, this formula is probably not valid (true in all states). The car model c is probably capable of running for 0 time units (e.g., no one turned the ignition key). In that case, the velocity is not positive. But in dL, you can simply use an implication to assume the initial velocity was positive:

$$v > 0 \rightarrow [c]\, v > 0$$

This formula now expresses that if the velocity is positive (initially), then after all runs of c is the velocity still positive. Transliteration into KeYmaera X with a car control model c reads as follows, using `[...]` for the diamond modality:

```
v>0 -> [c;]v>0
```

Proving it will not be possible before we define the hybrid systems model `c` as a hybrid program in the next chapter, because it very much depends on the specific car controller whether this formula is indeed true in all states.

Example 1.7 (Abstract diamond modalities)**.** Even if your favorite car model c cannot possibly be expected to always have a positive velocity at all times (just think of a traffic jam), it's quite plausible for it to reach a positive velocity at some point. In fact, otherwise you should park your car and take a walk instead. If c is a hybrid systems model for your car, the fact that it can run in some way to reach a positive velocity is expressed by a diamond modality :

$$\langle c \rangle\, v > 0$$

The transliteration in KeYmaera X is as follows using `<...>` for the diamond modality:

```
<c;>(v>0)
```

You are forgiven for misreading the above formula without parentheses even if `<c;>v>0` is understood just fine by KeYmaera X. Come to think of it, it may help cars obtain a positive speed if their acceleration a is large enough. So the following dL formula expresses that if the initial velocity is positive and the acceleration is at least 1 then the car model c can run to a state where the velocity is at least 5:

$$v > 0 \land a \geq 1 \rightarrow \langle c \rangle v \geq 5$$

Transliterating into KeYmaera X yields:

```
v>0 & a>=1 -> <c;> v>=5
```

Again a proof and the truth of this formula depends on the specifics of the car control model c which will be addressed in the next tutorial part (2).

1.4 Operator Precedence for Formulas

Operator precedence. Writing fully parenthesized formulas is extraordinarily tedious and quite unreadable on top of that. A merely syntactic but important convention are the operator binding precedences which determine how implicit parentheses are meant in case they are left out.

Important (Operator precedence for differential dynamic logic). To save parentheses, the notational conventions have unary operators (including $\neg$, quantifiers $\forall x, \exists x$, modalities $[a], \langle a \rangle$) bind more strongly than binary operators. We let $\land$ bind more strongly than $\lor$, which binds more strongly than $\rightarrow, \leftrightarrow$. Arithmetic operators $+, -, \cdot$ have the usual precedence and associate to the left. All logical operators associate to the right.

Consequences of operator precedences. These precedences imply that quantifiers and modal operators bind strongly, i.e., their scope only extends to the formula immediately after. So, $[a]P \wedge Q \equiv ([a]P) \wedge Q$ and $\forall x\, P \wedge Q \equiv (\forall x\, P) \wedge Q$ and $\forall x\, P \rightarrow Q \equiv (\forall x\, P) \rightarrow Q$. All logical operators associate to the right, most crucially $P \rightarrow Q \rightarrow R \equiv P \rightarrow (Q \rightarrow R)$. To avoid confusion, we do not adopt precedence conventions between $\rightarrow, \leftrightarrow$ but expect explicit parentheses. So $P \rightarrow Q \leftrightarrow R$ is illegal and explicit parentheses are required to distinguish $P \rightarrow (Q \leftrightarrow R)$ from $(P \rightarrow Q) \leftrightarrow R$. Likewise $P \leftrightarrow Q \rightarrow R$ is illegal and explicit parentheses are required to distinguish $P \leftrightarrow (Q \rightarrow R)$ from $(P \leftrightarrow Q) \rightarrow R$.

Example 1.8 (Redundant parentheses)**.** Formulas with too many parentheses quickly get unreadable. If the formula gets larger you will find yourself loosing time by counting parentheses. But, of course, it is critical to place parentheses whenever that is important for the formula to have the intended meaning.

The operator precedence implies that some parentheses can be left out from the above formula from Example 1.3 without changing its meaning, both in dL and in KeYmaera X:

$$v^2 > 0 \rightarrow (x \neq 0 \leftrightarrow xv > 0 \vee xv < 0)$$

Operator precedence rules can also be used to equivalently leave out several parentheses from Example 1.5 in dL and in KeYmaera X:

$$v^2 > 0 \rightarrow \forall x\, (x \neq 0 \rightarrow \exists c\, xvc^2 = 1 \vee \exists c\, xvc^2 = 1)$$

1.5 Optional: Function Symbols

Function symbols. You now saw the essential parts of differential dynamic logic formulas except the crucial part of hybrid programs. Depending on your interest, you may want to skip ahead right to the action with hybrid programs in the next tutorial part (2) and come back to the remaining optional parts of this tutorial part at a later point.

Function and predicate symbols can be useful to modularize your model. We first discuss the most common case of constant function symbols without arguments. A function symbol f needs to be applied to the correct number of arguments. When a function symbol f expects n arguments then $f(e1, \dots, en)$ is a term for terms $e1, \dots, en$. In the case this number of arguments or *arity* n is zero, f is a constant function symbol whose value does not depend on any arguments (there are none), but that still has a fixed real value, once and for all. For emphasis, the term is sometimes written `f()` to indicate that f is a constant function symbol with 0 arguments.

Example 1.9 (Constant function symbol)**.** Function symbols of no arguments can be helpful because it is clear without proof that they cannot change their (real) value during the evolution of your cyber-physical system. Suppose you want to use `T` for your systems' reaction time, which is 2 time units. Then you define its value as

```
Definitions
  Real T = 2;
End.
```

Suppose you want to use `T` for your systems' reaction time but do not need to give `T` any particular value. Then you declare it as

```
Definitions
  Real T;
End.
```

No matter whether you make `T` an interpreted function symbol with a specific value or an uninterpreted one, you can subsequently use it in your KeYmaera X question.

```
Definitions
  Real T;     /* reaction time constant */
End.
ProgramVariables
  Real x;     /* position variable */
  Real v;     /* velocity variable */
End.
Problem
  v>0 & T>0 -> \exists t (t>=0 & t<=T & x+v*t=x)
End.
```

Load this example in KeYmaera X and **Auto**. But notice how important the assumption `T>0` is when you have not given `T` a particular value such as `2`.

Note that you are strongly advised to declare variables and definitions and document their intention with comments. If you declare one, you have to declare them all. Program variables are declared within the `ProgramVariables` block, all other definitions within the `Definitions` block.

Arity. Function symbols with a positive arity, so a nonzero number of arguments can be defined as well. Some even come predefined as interpreted functions in KeYmaera X (namely `abs`, `min`, and `max`), but the majority is up for grabs by you.

Example 1.10 (Uninterpreted function symbol)**.** Suppose you frequently need the distance between two one-dimensional vectors. There are many different possible definitions for distance (especially in higher dimensions). So if you want to make your proof work for any such definition, you could work with an arity 2 function symbol, `dist`, capturing the abstract distance as an uninterpreted function symbol. That is, `dist` will be a real-valued function of 2 real-valued arguments but there's no way of knowing which function it is.

```
Definitions
  Real dist(Real x, Real y);
End.
ProgramVariables
  Real x;
  Real y;
  Real z;
End.
Problem
  dist(x,y) < 2 & dist(y,z) < 3 -> dist(x,z) < 5
End.
```

Now note how you have not assumed anything about the meaning of `dist`, so you will not be able to prove the above. That is why often you need to assume things about uninterpreted functions to make them useful. Or make them interpreted, which is what we consider next.

Example 1.11 (Interpreted function symbol)**.** Suppose you frequently need the distance between two one-dimensional vectors. You have a definition of distance in mind but may change your mind about it later because there are many notions (especially in higher dimensions). If you want to modularize and better structure your proof, you could work with an arity 2 function symbol, `dist`, that you define as a specific function, say the squared difference.

```
Definitions
  Real dist(Real x, Real y) = ((x-y)^2);
End.
ProgramVariables
  Real x;
  Real y;
  Real z;
End.
Problem
  dist(x,y) < 2 & dist(y,z) < 3 -> dist(x,z) < 5
End.
```

Be sure to enclose the right-hand side of the function definition in parentheses. Load this example in KeYmaera X and the proof will fail with a counterexample. Don't forget to click **Defs->Expand all definitions** to plug in the definition of `dist` at some point during your proof, e.g., right away. The reason is that the distance was defined as the squared distance. So either you change the definition of `dist` to the square root of the squared difference (yes this is more exciting in higher dimensions):

```
Definitions
  Real dist(Real x, Real y) = (((x-y)^2)^0.5);
End.
```

Or you leave the definition of `dist` untouched and, instead, change the property to prove by working with squares of the numbers:

```
Definitions
  Real dist(Real x, Real y) = ((x-y)^2);
End.
ProgramVariables
  Real x;
  Real y;
  Real z;
End.
Problem
  dist(x,y) < 2^2 & dist(y,z) < 3^2 -> dist(x,z) < 5^2
End.
```

The advantage of the former is that it matches geometric distance intuitions better. The advantage of the latter is that it avoids the technical nuisance complications associated with proving properties about square roots at the expense of remembering to use squares. Load this example in KeYmaera X and **Defs->Expand all definitions** then **Auto** proves it automatically for either model. Instead of expanding all definitions at once with **Defs->Expand all definitions**, you can also selectively click on some definitions to expand them

as needed. Delaying expansions may help keeping proofs more readable and focusing on only the detail that is presently relevant.

1.6 Optional: Predicate Symbols

Predicate symbols. Predicate symbols can be useful to modularize your model just like function symbols, but they give formulas, not terms. A predicate symbol p needs to be applied to the correct number of arguments. When a predicate symbol p expects n arguments then $p(e1, \dots, en)$ is a formula for terms $e1, \dots, en$. In the case this number of arguments or *arity* n is zero, f is a constant predicate symbol whose truth-value does not depend on any arguments (there are none), but that still has a fixed truth-value, once and for all. For emphasis, the formula is written `p()` to indicate that p is a constant predicate symbol with 0 arguments. Constant predicate symbols are not anywhere near as useful as constant function symbols, because they are either equivalent to the formula $true$ or to the formula $false$, we just don't know which one. So constant predicate symbols have less variations and also less use cases in your models than constant function symbols. For that reason, we will simply skip their explanation and go right for predicate symbols with arguments. Likewise, we will skip the explanation of uninterpreted predicate symbols, because, while useful, their role is very similar to uninterpreted function symbols and you can probably infer how to use them from the explanation of interpreted predicate symbols.

Example 1.12 (Interpreted predicate symbol)**.** Suppose you frequently need a safety notion determining whether two one-dimensional vectors are too close together. You have an appropriate definition of a safe separation in mind but may change your mind about it later because there are many notions (especially in higher dimensions). If you want to modularize and better structure and document your proof, you could work with an arity 2 predicate symbol, `sep`, that you define as a specific predicate, say the squared difference is at least 5.

```
Definitions
  Bool sep(Real x, Real y) <-> ((x-y)^2 > 5^2);
End.
ProgramVariables
  Real x;
  Real z;
End.
Problem
  sep(x, z) -> sep(z, x)
End.
```

Load this example in KeYmaera X and **Auto** but don't forget to click **Defs->Expand all definitions** at some point. In fact, you can also just expand definitions lazily by clicking on a top-level occurrence of `sep(x, z)` in the proof and it will rewrite itself using its definition.

If you later end up changing the notion of safe separation, then it suffices to change the definition of `sep` and reprove it without having to change all occurrences of `sep`. For example, if distance larger than 5 turns out to be needed, the following definition may help:

```
Definitions
```

```
  Bool sep(Real x, Real y) <-> ((x-y)^2 > 10^2);
End.
```

But if you end up changing the margin for what you call a safe distance a lot, it may be better to give it its own interpreted constant function symbol:

```
Definitions
  Real safesep = 10;
  Bool sep(Real x, Real y) <-> ((x-y)^2 > safesep^2);
End.
```

Notice how you can use one definition within another. Just don't try to build a chain of recursive occurrences or you will run into logical subtleties and find it impossible to prove by expanding them. Delaying expansions may help keeping proofs more readable and focusing on only the detail that is presently relevant.

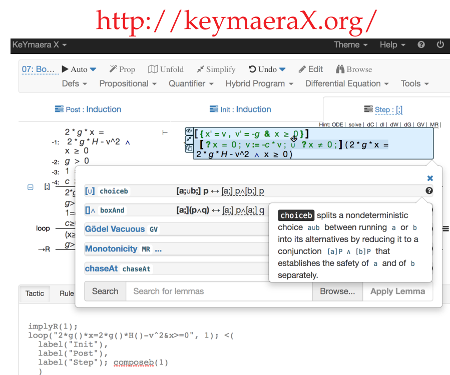
Chapter 2

Hybrid Programs in KeYmaera X

Welcome to the KeYmaera X Tutorial in which you will learn how to use the KeYmaera X aXiomatic Tactical Theorem Prover for Hybrid Systems from a pragmatic perspective.

KeYmaera X KeYmaera X is a theorem prover for *differential dynamic logic* (dL), a logic for specifying and verifying properties of *hybrid systems* with mixed discrete and continuous dynamics. This tutorial provides practical tool aspects and is complementary to the textbook *Logical Foundations of Cyber-Physical Systems*, in which provides comprehensive information on differential dynamic logic can be found. KeYmaera X is available at

http://keymaeraX.org/



Part Summary This part will give you an opportunity to learn how to write hybrid systems as hybrid programs for differential dynamic logic in KeYmaera X. Hybrid programs can be written within the box and diamond modalities of differential dynamic logic. They describe the operations that your hybrid system performs in a programming language. Besides having real-valued variables and nondeterministic operations to do justice to the uncertainties of the world around the system, the defining feature of hybrid programs are their differential equations that directly describe the continuous dynamics of your cyber-physical system.

Background This tutorial assumes that you have read or refer to the following chapters in the *Logical Foundations of Cyber-Physical Systems* textbook for background information on the principles as needed:

- Chapter 3: Choices & Control
- Chapter 2: Differential Equations & Domains
- Chapter 4: Safety & Contracts

2.1 Hybrid Program Statements

HP Statements. The modalities $[\alpha]$ and $\langle\alpha\rangle$ of differential dynamic logic dL described in the previous tutorial part (1) are desperately waiting for more information on what can be written down for α. In order to model cyber-physical systems models, α should have the mixed discrete-continuous dynamics of a *hybrid system*. Discrete dynamics comes up naturally from computation and decisions that change the system at an instant of time. Continuous dynamics comes up naturally, e.g., from continuous motion or other physical processes.

Differential dynamic logic supports *hybrid programs* that are a programming language for hybrid systems. Besides supporting real-valued variables to reflect real positions or velocities of cyber-physical systems, the constituent features of hybrid programs (*HPs*) are assignments for discrete computation and differential equations for continuous behavior. Another important feature of hybrid programs is the direct support for *nondeterminism*, because the real world and behavior of other agents in the environment often cause so many uncertainties that it is imperative to consider multiple possible behaviors. In fact, even the controller of your own cyber-physical system may exhibit multiple slightly different behaviors due to tolerances. And even if it does not, it may be invaluable for complexity management to develop nondeterministic models regardless, as you can learn about in the textbook.

Without further ado, here is the syntax of hybrid programs, which is also summarized succinctly on the KeYmaera X Cheat Sheet.

Statements and effects of Hybrid Programs (HPs) and Hybrid Games (HGs).

HP	KeYmaera X	Operation	Effect
$x := e$	`x:=e;`	discrete assignment	assigns value of term e to variable x
$x := *$	`x:=*;`	nondet. assign	assigns any real value to variable x
$x' = f(x) \,\&\, Q$	`{x'=f(x)&Q}`	continuous evolution	evolve along differential equation $x' = f(x)$ within evolution domain Q for any duration
$?Q$	`?Q;`	test	check first-order formula Q at current state
$a; b$	`a; b`	seq. composition	HP b starts after HP a finishes
$a \cup b$	`a ++ b`	nondet. choice	choice between alternatives HP a or HP b
a^*	`{a}*`	nondet. repetition	repeats HP a n-times for any $n \in \mathbb{N}$
$\mathsf{if}(Q)\, a\, \mathsf{else}\, b$	`if(Q) {a} else {b}`	if-then-else	HP a runs if Q, otherwise b runs
a	`a;`	atomic program	run program symbol or subprogram a
a^{d}	`{a}^@`	dual game	HG: hybrid game duality operator
$a \cap b$	`a -- b`	demonic choice	HG: Demon's choice between HG a or b

Unary operators (*) bind stronger than binary operators and ; binds stronger than ∪. Braces `{...}` group programs and `;` terminates atomic programs.

Operator precedence. Unary operators (including *) bind more strongly than binary operators. We let ; bind more strongly than ∪. Just like logical operators but unlike arithmetic operators, all program operators associate to the right. The primary difference between the mathematical notation for hybrid programs in dL compared to ASCII notation in KeYmaera X is the `;` termination of statements in KeYmaera X, that ∪ is written `++`, and that braces `{..}` are used instead of round parentheses for grouping programs and are *required* around differential equations and nondeterministic repetitions. Hybrid games also support duality and demonic choice but are explained elsewhere.

KeYmaera X notation. Note a slight notational subtlety with the `;` for sequential compositions: it is not needed if your prior statement ends in a `;` anyhow. For example `x:=5;x:=x+1;` is a perfectly reasonable sequential composition of two assignments, there is no need to write `x:=5;;x:=x+1;`. But if you use braces for HP blocks, you will usually find it more readable to have a ; for sequential composition as in `{x:=5;};{x:=x+1;}` than to leave them out `{x:=5;}{x:=x+1;}` although both versions parse the same way.

Primed variables. Note that, during the differential equation `{x'=f(x) & Q}`, the differential variable x' with a prime denotes the time-derivative of x. Outside differential equations, however, you can think of a primed variable as just another variable. The intrinsic link between x and x' with x' equaling the time-derivative of x is important during a differential equation. But there are no time-derivatives outside differential equations.

Differential equation system notation. Besides braces around them, differential equation systems are simply written with commas, e.g., the differential equation system $x' = v, v' = a, t' = 1 \,\&\, t \leq 5$ is written as follows in KeYmaera X:

```
{x'=v, v'=a, t'=1&t<=5}
```

The evolution domain constraint Q of a differential equation $x' = f(x) \,\&\, Q$ cannot be left at any time during the evolution along this differential equation: Q has to be true at all times along the solution. If Q is not even true in the initial state, no solutions of $x' = f(x) \,\&\, Q$ exist. Evolution domain constraints are optional, however, and can be left out instead of explicitly writing the formula `true`. For example $x' = v, v' = a, t' = 1$ is equivalent to $x' = v, v' = a, t' = 1 \,\&\, \mathit{true}$ and simply written as follows in KeYmaera X:

```
{x'=v, v'=a, t'=1}
```

Note that there is a huge difference between the differential equation system

```
{x'=v, v'=a}
```

and this sequential composition of two differential equations:

```
{x'=v};{v'=a}
```

In the former, the velocity `v` used in the ODE `x'=v` is already changing according to `v'=a`, which is why this gives accelerated motion. In the latter, the two successive differential equations are unrelated and may be followed for entirely different durations, where first

`x` changes according to the constant initial velocity `v` along `x'=v` and then, subsequently, `v` is changing according to the constant acceleration `a` along `v'=a`. This is why braces are required around differential equation systems as a visual cue to make it easier to tell both situations apart.

Example 2.1 (Simplistic car controller)**.** As a first example of a hybrid program, consider this simplistic car controller:

$$\Big(\big((?v < 4;\, a := a + 1) \cup a := -b\big);\; \{x' = v, v' = a\}\Big)^*$$

This HP repeatedly (as indicated by the repetition * at the end) runs its discrete control followed by (after ;) a differential equation. The differential equation $\{x' = v, v' = a\}$ indicates that the car's position x is changing with a time-derivative x' that equals the car's velocity v while the velocity is changing with a time-derivative v' that equals the acceleration for any amount of time. In every round of the repetition, the discrete controller first has two choices, to increase the acceleration a by assigning $a := a + 1$ or to reset a to $-b$ by $a := -b$ for braking. The choice between both subprograms is nondeterministic ($\cup$) except that the left choice first needs to pass a test $?v < 4$ requiring that the velocity v is less than 4. If the velocity is, indeed, less than 4 then both choices $a := a + 1$ and $a := -b$ are possible, otherwise only the braking choice is possible, because the left choice would fail its test. Failed attempts to run an HP are discarded, because they were not successful. So only runs that pass all tests are relevant.

This HP is transliterated into KeYmaera X as follows using braces instead of parentheses for grouping HPs:

```
{{{?v<4;a:=a+1;} ++ a:=-b;}; {x'=v,v'=a}}*
```

Example 2.2 (Acrophobic bouncing ball)**.** Recall "Quantum", the acrophobic bouncing ball that always wants to stay above ground but below the initial height, because he's afraid of heights. Here is the model from Chapter 4.

$$\big(\{x' = v, v' = -g \,\&\, x \geq 0\};\; \mathsf{if}(x = 0)\, v := -cv\big)^*$$

Repeatedly (as indicated by the * at the end), the bouncing ball will first fly through the air and fall in gravity according to a differential equation that has the altitude x changing with a time-derivative of v which is changing with a time-derivative equaling negative gravity $-g$. A careful observer will miss the if-then statement in the syntax of the hybrid programs. But if-then-elses are easily definable by nondeterministic choices with tests, giving:

$$\big(\{x' = v, v' = -g \,\&\, x \geq 0\};\; ((?x = 0; v := -cv) \cup ?x \neq 0)\big)^*$$

Transliterating this HP to KeYmaera X results in:

```
{{x'=v,v'=-g&x>=0}; {{?x=0;v:=-c*v;} ++ ?x!=0;}}*
```

But KeYmaera X also directly supports if-then-else statements with braces around its subprograms, so you could equivalently write:

```
{{x'=v,v'=-g&x>=0}; if (x=0) {v:=-c*v;}}*
```

2.2 dL Modalities with Hybrid Programs

dL Modalities with HPs. The box modality $[\alpha]$ can be applied to any dL formula P, giving a new dL formula $[\alpha]P$, which expresses that formula P is true after all runs of α. Likewise, the diamond modality $\langle\alpha\rangle$ can be applied to any dL formula P, giving a new dL formula $\langle\alpha\rangle P$, which expresses that P is true after some run of α (at least one run). Now that you understand the structure of hybrid programs α, you can write them right in the modalities of differential dynamic logic to specify (and ultimately verify) their correctness.

Example 2.3 (Simplistic car control goes forward)**.** Coming back to the hybrid program for the simplistic car controller from Example 2.1 here is a differential dynamic logic formula conjecturing that it never moves backward:

$$v \geq 0 \rightarrow [\big(((?v < 4;\ a := a + 1) \cup a := -b);\ \{x' = v, v' = a\}\big)^*]\, v \geq 0$$

If the velocity v is initially nonnegative, then it will always be nonnegative when following the simple car control model. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
v>=0 -> [{{{?v<4;a:=a+1;} ++ a:=-b;}; {x'=v,v'=a}}*] v>=0
```

That is a wonderful conjecture but sadly not true in all states. In fact, it is even false in almost all states you can think of. It is a good exercise to find a fix for the above conjecture and then try and prove it in KeYmaera X by **Auto**.

Example 2.4 (Fast car)**.** Consider a car with a fast gear a and a slow gear b that can switch arbitrarily (and instantly) between the two. The following differential dynamic logic formula conjectures that this car will always have a nonnegative position:

$$x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big((v := a \cup v := b);\ \{x' = v\}\big)^*]\, x \geq 0$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>=0 & a>0 & b>0 -> [{{v:=a; ++ v:=b;}; {x'=v}}*] x>=0
```

Load this example in KeYmaera X and prove it automatically by clicking **Auto**. If you have configured KeYmaera X correctly it should prove immediately.

2.3 Operator Precedence for Programs

Operator precedence. Writing fully parenthesized programs is just as tedious as writing fully parenthesized formulas and just as unreadable. A merely syntactic but important convention are the operator binding precedences which determine how implicit parentheses are meant in case they are left out.

Important (Operator precedence for differential dynamic logic)*.* To save parentheses, the notational conventions have unary operators (including $\neg$, quantifiers $\forall x, \exists x$, modalities $[a], \langle a\rangle$, and repetition *) bind more strongly than binary operators. We let $\land$ bind more strongly than $\lor$, which binds more strongly than $\rightarrow, \leftrightarrow$, and let ; bind more strongly than $\cup$. Arithmetic operators $+, -, \cdot, /$ have the usual precedence and associate to the left, powers associate to the right. All logical and program operators associate to the right.

Consequences of operator precedence. These precedences imply that quantifiers and modal operators bind strongly, i.e., their scope only extends to the formula immediately after. So, $[a]P \land Q \equiv ([a]P) \land Q$ and $\forall x\, P \land Q \equiv (\forall x\, P) \land Q$ and $\forall x\, P \to Q \equiv (\forall x\, P) \to Q$. They imply $a; b \cup c \equiv (a; b) \cup c$ and $a \cup b; c \equiv a \cup (b; c)$ and $a; b^* \equiv a; (b)^*$ like in regular expressions. All logical and program operators associate to the right, most crucially $P \to Q \to R \equiv P \to (Q \to R)$. To avoid confusion, we do not adopt precedence conventions between $\to, \leftrightarrow$ but expect explicit parentheses. So $P \to Q \leftrightarrow R$ is illegal and explicit parentheses are required to distinguish $P \to (Q \leftrightarrow R)$ from $(P \to Q) \leftrightarrow R$. Likewise $P \leftrightarrow Q \to R$ is illegal and explicit parentheses are required to distinguish $P \leftrightarrow (Q \to R)$ from $(P \leftrightarrow Q) \to R$.

Example 2.5 (Redundant parentheses)**.** Some parentheses can be removed from the HP for the bouncing ball from Example 2.2 without changing its meaning:

$$\big(\{x' = v, v' = -g \,\&\, x \geq 0\};\ (?x = 0; v := -cv \cup ?x \neq 0)\big)^*$$

Or, equivalently in KeYmaera X:

```
{{x'=v,v'=-g&x>=0}; {?x=0;v:=-c*v; ++ ?x!=0;}}*
```

If you have already read Chapter 7: Loops & Invariants, you can also try to identify a loop invariant and attach it after the * with `@invariant(.....)`. Then try to see if you can prove the postcondition $0 \leq x \land x \leq H$ for the initial height H after identifying a suitable precondition that makes such a dL formula true in all states.

Parentheses can be elided in the simple car HP in the same way that they can be elided in the dL formula from Example 2.3:

$$v \geq 0 \to [\big((?v < 4;\ a := a + 1 \cup a := -b);\ \{x' = v, v' = a\}\big)^*]\, v \geq 0$$

Or, equivalently in KeYmaera X:

```
v>=0 -> [{{?v<4;a:=a+1; ++ a:=-b;}; {x'=v,v'=a}}*] v>=0
```

Have you found a way to fix this model yet, to make such a conjecture true?

2.4 Optional: Nondeterministic Assignments

Example 2.6 (Faster car)**.** Your fast car model from 2.4 had two gears, a slow one and a fast one. You could add a third choice to the model for an intermediate gear. Or add 4 choices to end up with 6 gears. But what if your car is supposed to have even more gears than that. How? Real cars use Continuously Variable Transmission but car models just need a nondeterministic assignment. Consider a car with a gear a that can change to an arbitrary real number, including really fast gears. The following differential dynamic logic formula conjectures that this car will always have a nonnegative position:

$$x \geq 0 \to [\big(v := *;\ \{x' = v\}\big)^*]\, x \geq 0$$

Every time around the loop, it can choose an arbitrary new real number (or the same number from last round) for the velocity and then follow the differential equation $x' = v$ for some amount of time. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>=0 -> [{v:=*; {x'=v}}*] x>=0
```

Load this example in KeYmaera X and try and prove it.

After you have tried proving this dL formula for some time, you probably came back here disappointed that you were unable to do so. There is a very compelling reason why you cannot and should not have been successful proving this formula: it is not valid. If, ever, the gear is chosen to an arbitrary real number such that the resulting velocity is negative, then following that differential equation for sufficiently long will make any position x negative, thereby falsifying the conjecture that $x \geq 0$ is always true. How can this be fixed?

If you add a subsequent test for nonnegativity right after the arbitrary real choice of velocity, then no attempt of choosing a negative velocity will succeed, so only nonnegative velocities remain.

$$x \geq 0 \rightarrow [\big(v := *; ?v \geq 0;\ \{x' = v\}\big)^*]\, x \geq 0$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>=0 -> [{v:=*;?v>=0; {x'=v}}*] x>=0
```

Load this example in KeYmaera X and use **Auto** to prove it.

2.5 Optional: Program Symbols for Subprograms

Subprogram symbols. If you are not yet interested in structuring larger models into smaller pieces, you can skip this section and return to it at a later point. Function and predicate symbols were already described in the previous tutorial part (1). Refer back to 1.5 and 1.6 to see how they work. What remains is the question how program symbols can be defined that are used as subprograms. They again come in two forms, interpreted and uninterpreted program symbols. But the interpreted program symbols are going to be much more useful for your models than uninterpreted program symbols, so this tutorial will focus only on those. Interpreted program symbols enable you to define what specific subprogram they refer to, which is more useful in models.

Example 2.7 (Interpreted program symbol)**.** Recall Example 2.4.

$$x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big((v := a \cup v := b);\ \{x' = v\}\big)^*]\, x \geq 0$$

and its corresponding transliteration to KeYmaera X:

```
x>=0 & a>0 & b>0 -> [{{v:=a; ++ v:=b;}; {x'=v}}*] x>=0
```

While there is nothing wrong with this representation, things can get more untidy for larger models. So here's how you can define an interpreted program symbol `ctrl` for the controller and another interpreted program symbol `motion` for its differential equation:

```
Definitions
  HP ctrl   ::= {v:=a; ++ v:=b;};   /* discrete control subprogram */
  HP motion ::= {{x'=v}};           /* ODE subprogram */
End.
ProgramVariables
  Real x;    /* position */
  Real v;    /* velocity */
```

```
End.
Problem
  x>=0 & a>0 & b>0 -> [{ctrl; motion;}*] x>=0
End.
```

Don't forget to terminate every use of a program symbols with a ; just like assignments. For example `ctrl;` refers to the controller {`v:=a; ++ v:=b;`} and `plant;` refers to the ODE `{x'=v}`. Any use of the subprogram `ctrl;` can be thought of as synonymous to the mention of its expansion {`v:=a; ++ v:=b;`}. Likewise does any mention of the `plant;` subprogram stand for the ODE `{x'=v}`. Every HP definition needs to be wrapped in curly braces, so you may need double curly braces when an HP is noting other than a single differential equation system. Load this example in KeYmaera X and use **Defs->Expand all definitions** followed by **Auto**. The expansion of the definition will uniformly replace all occurrences of `ctrl;` by {`v:=a; ++ v:=b;`} and all occurrences of `plant;` by `{x'=v}`. It can be helpful to delay such expansions till later in a proof.

Example 2.8 (Interpreted definitions)**.** Continuing the above example, you can also use a mix of hybrid program definitions, predicate definitions and function definitions in your model.

```
Definitions
  Real a;              /* any uninterpreted velocity */
  Real b;              /* any uninterpreted velocity */
  Real plim = 2;       /* fixed lower position limit 2 */
  Bool init(Real x) <-> (x=plim());  /* initial condition */
  Bool consts       <-> (a>0 & b>0); /* constant assumptions */
  Bool safe(Real x) <-> (x>=plim()); /* safety */
  HP ctrl    ::= {v:=a; ++ v:=b;};     /* discrete control subprogram */
  HP motion  ::= {{x'=v}};             /* ODE subprogram */
End.
ProgramVariables
  Real x;     /* position */
  Real v;     /* velocity */
End.
Problem
  init(x) & consts() -> [{ctrl; motion;}*] safe(x)
End.
```

Note how most symbols have a definition, so they mean exactly one thing. For example `plim` is defined to be 2. But some function symbols do not have a definition, notably `a`, `b`, which means that they could have any arbitrary fixed real value. The `consts()` constant predicate symbol used in the problem statement assumes both `a` and `b` are positive.

The hybrid program `ctrl` is defined to be the discrete control program `v:=a; ++ v:=b;` while the HP `motion` is defined to be the differential equation system {`x'=v`}. The predicate `init(Real x)` of one real argument `x` is defined to be the equality comparison `x=plim()` of its argument to the value of `plim` which, in turn, is defined to be 2.

Other function or predicate symbols can be used in definitions, but program variables need to be passed in explicitly as arguments. While proving this example, don't forget to **Defs->Expand all definitions** at some point. Again, it may help to wait with expanding the definitions until the particular definition becomes important.

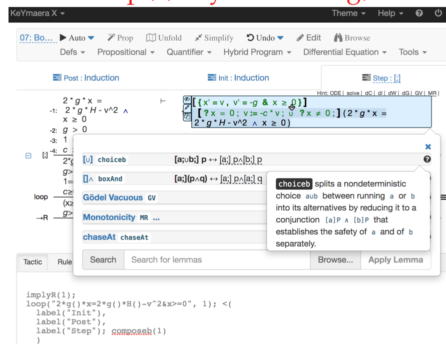
Chapter 3

Proofs in KeYmaera X

Welcome to the KeYmaera X Tutorial in which you will learn how to use the KeYmaera X aXiomatic Tactical Theorem Prover for Hybrid Systems from a pragmatic perspective.

KeYmaera X KeYmaera X is a theorem prover for *differential dynamic logic* (dL), a logic for specifying and verifying properties of *hybrid systems* with mixed discrete and continuous dynamics. This tutorial provides practical tool aspects and is complementary to the textbook *Logical Foundations of Cyber-Physical Systems*, in which provides comprehensive information on differential dynamic logic can be found. KeYmaera X is available at

http://keymaeraX.org/



Part Summary This part will explore how explicit proofs in differential dynamic logic can be conducted in KeYmaera X. After you have seen how differential dynamic logic and its hybrid programs can be written in KeYmaera X for specification purposes, you will now explore how to prove them for verification purposes.

This tutorial part will open up the box and allow you to look inside the automatic proof search that you have explored so far. Proof search automation is extraordinarily helpful. But there is always a cyber-physical system that is more complicated than any existing automatic verification technique. In theorem provers, however, you can overcome such limits by following the proofs and use your system expertise to help out when the automation gets stuck. Before you face systems that are so complicated that automatic proofs are impossible, however, you should practice with proofs on simpler examples to

understand how this works. Understanding how to prove simpler systems will be a perfect preparation for more complicated systems and will also enable you to better predict what proof automation will do.

Background This tutorial assumes that you have read or refer to the following chapters in the *Logical Foundations of Cyber-Physical Systems* textbook for background information on the principles as needed:

- Chapter 5: Dynamical Systems & Dynamic Axioms
- Chapter 6: Truth & Proof
- Chapter 7: Loops & Invariants

Video KeYmaera X Tutorial 01: Usage Overview

3.1 Propositional Proofs

Propositional proofs. The fully automatic proofs in this tutorial so far have been fun. But they were not very insightful for understanding how proofs work. Especially very complicated cyber-physical systems that are out of reach for full automation benefit from mixed automatic and interactive proofs. Doing such proofs can also be exceedingly helpful to debug controllers that are still broken and find out how they need to be fixed. Yet, before jumping right into the most complex systems it's best if you first understand the principles on simpler examples.

Example 3.1 (Propositional logic proofs)**.** Consider a simple formula using only propositional logical connectives:

$$v > 0 \land (x > 0 \land v > 0 \rightarrow xv > 0) \rightarrow (x > 0 \rightarrow xv > 0)$$

If the velocity v is positive and if it is the case that positive position x and positive velocity v imply a positive product xv, then a positive position x implies a positive product xv. In fact, the particular arithmetic is even unimportant, because the logical argument justifying this is independent of the specific terms, so you can do an entirely propositional proof. Transliterating the formula into KeYmaera X gives:

```
v>0 & (x>0 & v>0 -> x*v>0) -> (x>0 -> x*v>0)
```

When you load this model in KeYmaera X you can, of course, prove it by **Auto** for full automation but also with **Prop** which uses dedicated propositional logic proof automation. **Prop** can be much faster than **Auto** but will only succeed if propositional reasoning suffices.

Alternatively you can also conduct an interactive proof. Just **left-click** on the formula to apply the most suitable axiom or proof rule. If you are unsure which rule to apply, a **right-click** will bring up a context menu showing you a list of the most promising applicable axioms and proof rules. (Although some operating systems do not support right-clicks and require an option-click or alt-click instead).

For example the →R rule is a great proof rule to start your proof with. It proves an implication $P \to Q$ by assuming its left-hand side P in the list of all assumptions before $\vdash$ and proceeds to prove its right-hand side Q.

$$\text{(}\to\text{R)}\ \frac{\Gamma, P \vdash Q, \Delta}{\Gamma \vdash P \to Q, \Delta}$$

Why don't you try to find a proof yourself this way? If you want to look at your proof at any point, click the [+] button left of the proof rule bar.

Important (Sequent calculus). In a *sequent* $\Gamma \vdash \Delta$ all assumptions are gathered in the set of formulas Γ, called *antecedent*. The set of formulas Δ out of which at least one needs to be shown from Γ is called *succedent*. Sequents are a normal form used in proofs to organize all available assumptions left of the $\vdash$ turnstile and keep what needs to be shown on the right. It is enough to show one of the formulas in Δ or their disjunction from the conjunction of assumptions in Γ to prove a sequent.

A *proof rule* in sequent calculus has the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \dots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Proof rules in sequent calculus are used bottom-up, because you use them to decompose the goal with your desired conclusion $\Gamma \vdash \Delta$ at the bottom to the remaining subgoals with the premises $\Gamma_1 \vdash \Delta_1$ and ... and $\Gamma_n \vdash \Delta_n$ at the top. Think of using proof rules bottom up on your The *sound* proof rules of dL make sure that the conclusion $\Gamma \vdash \Delta$ at the bottom is indeed valid, so true in all states, if you found a proof for all premises $\Gamma_i \vdash \Delta_i$ for all i at the top. More details about the dL sequent calculus and its proof rules are in Chapter 6: Truth & Proof.

A succinct summary of proof rules is on the KeYmaera X Cheat Sheet.

Example 3.2 (Propositional proofs)**.** There are many different proofs for the formula in 3.1. Here's the proof I found:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{v>0,x>0 \vdash x>0}\text{id} \quad \dfrac{*}{v>0,x>0 \vdash v>0}\text{id}}{v>0,x>0 \vdash x>0 \land v>0}\land\text{R} \quad \dfrac{*}{v>0,x>0,xv>0 \vdash xv>0}\text{id}}{v>0,x>0 \land v>0 \to xv>0, x>0 \vdash xv>0}\to\text{L}}{v>0, x>0 \land v>0 \to xv>0 \vdash x>0 \to xv>0}\to\text{R}}{v>0 \land (x>0 \land v>0 \to xv>0) \vdash x>0 \to xv>0}\land\text{L}}{\vdash v>0 \land (x>0 \land v>0 \to xv>0) \to (x>0 \to xv>0)}\to\text{R}$$

Example 3.3 (Propositional proof tactics)**.** Here's the same proof of the same formula as in 3.2, now written as a Bellerophon tactic:

```
implyR(1) ; andL(-1) ; implyR(1) ; implyL(-2) ; <(
  andR(2) ; <(
    id,
    id
  ),
  id
)
```

When you switch to the **Tactic** tab you see your tactic and can paste this tactic instead and re-run the proof to compare. At this time, there is no need to understand the Bellerophon tactics, because a later part of this tutorial will cover them. Briefly, though, `implyR(1)` indicates that the implication proof rule on the right $\to$R is used at the first formula in the succedent after the $\vdash$ turnstile. Then subsequently, so after the `;` operator, the conjunction proof rule on the left $\wedge$L is used at the first formula in the resulting antecedent. The $<$ operator indicates branching of the proof and `id` is the identity proof rule closing proofs for sequents of the form $\Gamma, P \vdash P, \Delta$ whose assumptions directly equal one of the succedent formulas to show. Another way to read this explicit proof tactic off of the sequent proof is to start at the bottom and separate proof steps by `;`, indicating branching by the $<$ branching operator.

The above tactic is fairly short and explicitly indicates the integer position of the formulas where the tactics are applied. But is sometimes hard to read and stops working when the problem is changed. The following equivalent tactic explicitly indicates the formulas that the tactics are applied to on the left-hand side antecedent (by `'L`) or on the right-hand side succedent (by `'R`). It also labels branches to be easier to read.

```
implyR('R=="v>0&(x>0&v>0->x*v>0)->x>0->x*v>0");
andL('L=="v>0&(x>0&v>0->x*v>0)");
implyR('R=="x>0->x*v>0");
implyL('L=="x>0&v>0->x*v>0"); <(
  "x>0&v>0": andR('R=="x>0&v>0"); <(
      "x>0": id,
      "v>0": id
    ),
  "x*v>0": id
)
```

These were explicit tactics that directly instructs KeYmaera X which proof step to take in succession. Of course, a much shorter Bellerophon tactic would have worked just as well that searches for a propositional proof:

```
prop
```

A Bellerophon tactic that just calls full proof automation also suffices although it may be slower than `prop`:

```
auto
```

You can also perform all the canonical decompositions of propositional *and* program structure that do not branch or cause difficult decisions by using the **Unfold** button.

The advantage of these more searchy proof tactics is that they are easier to write and generalize better when the system changes. The downside is that they are not very explicit and may run into scalability limits for complicated systems. A mix of explicit and searchy tactics can be a good idea to combine the best of both worlds. Start with an explicit proof to give it some structure and then bottom-out in searchy proofs and finally full proof automation.

3.2 Quantifier Proofs

Example 3.4 (Quantifiers by real arithmetic quantifier elimination)**.** The following first-order formula can be proved by quantifier elimination in real arithmetic:

$$\exists x\,(x \geq 0 \land x^3 < x^2)$$

Transliterating into KeYmaera X mostly expands the quantifier into LaTeX notation:

```
\exists x (x>=0 & x^3<x^2)
```

Load this example in KeYmaera X and use **Tools->Real Arithmetic** to invoke quantifier elimination in real arithmetic to prove it.

Example 3.5 (Instantiating existential quantifiers in the succedent)**.** The proof for the example in 3.4 succeeds by real arithmetic quantifier elimination. But you neither learned a lot about why that property is true from such an automatic proof, nor was the proof as fast as it could have been. It is easy to find a number whose cube is less than its square that is also positive, for example 0.5. Load this example in KeYmaera X and **left-click** or **right-click** to choose the rule $\exists$R, click on the blue or red term θ in the rule, and enter the term `0.5` as witness. This performs the following proof step

$$\exists\mathrm{R}\ \frac{\vdash 0.5 \geq 0 \land 0.5^3 < 0.5^2}{\vdash \exists x\,(x \geq 0 \land x^3 < x^2)}$$

Ironically, you will still use **Tools->Real Arithmetic** to prove the resulting arithmetic in the remaining premise, but it's much faster now that $0.5 \geq 0 \land 0.5^3 < 0.5^2$ merely evaluates concrete numbers, which is much easier for KeYmaera X than to find existence of such a witness for itself. The resulting Bellerophon proof tactic is:

```
existsR("0.5", 1) ; QE
```

It specifies the term `0.5` to be used for instantiating the existential quantifier of the first formula in the succedent on the right hand side after the $\vdash$ turnstile, and then to use quantifier elimination. Now even if there are no quantifiers to speak of anymore—after all you've eliminated the $\exists$ by instantiating it cleverly—it's still the quantifier elimination tactic `QE` that can handle any remaining real arithmetic.

Giving explicit position numbers such as the formula at succedent position 1 in the above proof can be unreadable and cause stop working when the problem changes. A more readable and robust proof indicates the given formula on the right hand side (in the succedent):

```
existsR("0.5", 'R=="\exists x (x>=0&x^3 < x^2)")
```

Example 3.6 (More quantifiers by real arithmetic quantifier elimination)**.** The following first-order formula can be proved by quantifier elimination in real arithmetic:

$$\forall x\,(x \geq 0 \rightarrow \exists y\,(x = y^2)) \rightarrow (ab > 0 \rightarrow \exists y\,(ab + 1 = y^2))$$

Transliterating into KeYmaera X mostly expands the quantifier into LaTeX notation:

```
\forall x (x>=0->\exists y (x=y^2)) -> (a*b>0->\exists y (a*b+1=y^2))
```

Load this example in KeYmaera X and use **Tools->Real Arithmetic** to invoke quantifier elimination in real arithmetic to prove it. Can you find a direct proof not using quantifier elimination?

Example 3.7 (Instantiating universal quantifiers in the antecedent)**.** The proof for the example in 3.6 succeeds by real arithmetic quantifier elimination. But there is a much faster and more insightful proof. Its key insight, after using the $\rightarrow$R rule to isolate the universal quantifier for x in the antecedent is to instantiate it by the $\forall$L rule, where you click on the θ term in the rule and type in the term you are interested in, which is $ab + 1$. Then the remaining existentially quantified formulas are identical and prove by rule id, leaving only a fairly easy arithmetic question that $ab > 0$ implies $ab + 1 \geq 0$, which it obviously does.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{ab > 0 \vdash ab+1 \geq 0}\,\mathbb{R}}{ab > 0 \vdash ab+1 \geq 0, \exists y\,(ab+1 = y^2)}\,\text{WR} \qquad \dfrac{*}{\exists y\,(ab+1 = y^2), ab > 0 \vdash \exists y\,(ab+1 = y^2)}\,\text{id}}{ab+1 \geq 0 \rightarrow \exists y\,(ab+1 = y^2), ab > 0 \vdash \exists y\,(ab+1 = y^2)}\,\rightarrow\text{L}}{ab+1 \geq 0 \rightarrow \exists y\,(ab+1 = y^2) \vdash ab > 0 \rightarrow \exists y\,(ab+1 = y^2)}\,\rightarrow\text{R}}{\forall x\,(x \geq 0 \rightarrow \exists y\,(x = y^2)) \vdash ab > 0 \rightarrow \exists y\,(ab+1 = y^2)}\,\forall\text{L}}{\vdash \forall x\,(x \geq 0 \rightarrow \exists y\,(x = y^2)) \rightarrow (ab > 0 \rightarrow \exists y\,(ab+1 = y^2))}\,\rightarrow\text{R}$$

Notice how this instantiation of the universal quantifier in the antecedent leads to a more clever proof than trying to instantiate the existential quantifier in the succedent, because the witness for y would to be chosen as the somewhat unwieldy expression $\sqrt{ab+1}$, which needs to be transformed to `(a*b+1)^0.5` to even become a term. And even then, you can only use this term after you make sure that this square is well-defined. Here is the same proof written as a Bellerophon tactic:

```
implyR(1) ; allL("a*b+1", -1) ; implyR(1) ; implyL(-1) ; <(
  hideR(1=="\exists y a*b+1=y^2") ; QE,
  id
)
```

Another way to read this explicit proof tactic off of the sequent proof is to start at the bottom and separate proof steps by `;`, indicating branching by the < branching operator. The instance to use is indicated as an argument to the tactic `allL` along with the position of the formula, the first antecedent formula. The weakening rule WR is written as `hideR` in KeYmaera X and expects a position argument. For readability purposes, it is a good idea to indicate the formula that it weakens at that position. Overall, you have to admit that this Bellerophon tactic tells you more about the idea behind the proof than an equally successful automatic search tactic:

```
auto
```

The other reason why more interactive proofs can help is that the enable you to scale to much more complex cyber-physical systems that are still out of reach for fully automatic CPS verification. But it is good to practice on smaller systems that are still perfectly within reach to make sure you are not overwhelmed when the time comes that you've exceeded automation capabilities.

Clever real arithmetic simplifications. More clever techniques for simplifying the complexity of real arithmetic questions can be extraordinarily impactful. They are explained in Chapter 6: Truth & Proof.

3.3 Dynamics Proofs

Dynamic proofs. Proving dynamic modalities with hybrid programs in differential dynamic logic mostly proceeds by using the appropriate dynamic axioms. These are discussed in Chapter 5: Dynamical Systems & Dynamic Axioms in detail and summarized succinctly on the KeYmaera X Cheat Sheet.

Example 3.8 (Fast car)**.** Remember the fast car from 2.4? It already had a fully automatic proof for the following differential dynamic logic formula saying that the car with a slow and fast gear will always have a nonnegative velocity:

$$x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big((v := a \cup v := b);\ \{x' = v\}\big)^*]\, x \geq 0$$

Even if it had an automatic proof, you will best understand it by also conducting a manual proof. Since loops are the topic of Chapter 7: Loops & Invariants, we simply drop the loop for now and consider the following simpler dL formula instead:

$$x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big(v := a \cup v := b\big);\ \{x' = v\}]\, x \geq 0$$

A proof of this dL formula without a loop will not imply the original formula with a loop. But the loop could have run exactly once, so the loopy formula can only be valid if the one without the loop is. Remember that as a good test. Here is the same differential dynamic logic formula transliterated to KeYmaera X, without a loop:

```
x>=0 & a>0 & b>0 -> [{v:=a; ++ v:=b;}; {x'=v}] x>=0
```

Load this example in KeYmaera X and resist the temptation to just click **Auto**, because that would prove it automatically and you would not learn anything new. Instead follow the following steps. Use →R to move the assumption to the left-hand side of the turnstile. Then split the sequential composition using the axiom [;] by either a **left-click** or a **right-click** selecting axiom [;]. Then use axiom [∪] followed by rule ∧R to split the proof into the case of $v := a$ and the case of $v := b$. Then use axiom [:=] to substitute in the assignment into the differential equation and solve it with **right-click** via axiom ['] and eliminate the quantifiers in the resulting arithmetic by $\mathbb{R}$ via **Tools->Real Arithmetic**. Proceed similarly on the other remaining proof branch, giving you the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x \geq 0 \land a > 0 \land b > 0 \vdash \forall t\,(t \geq 0 \rightarrow x + at \geq 0)}\mathbb{R}}{x \geq 0 \land a > 0 \land b > 0 \vdash [x' = a]\, x \geq 0}[']}{x \geq 0 \land a > 0 \land b > 0 \vdash [v := a][x' = v]\, x \geq 0}[:=] \quad \dfrac{\dfrac{\dfrac{*}{x \geq 0 \land a > 0 \land b > 0 \vdash \forall t\,(t \geq 0 \rightarrow x + bt \geq 0)}\mathbb{R}}{x \geq 0 \land a > 0 \land b > 0 \vdash [x' = b]\, x \geq 0}[']}{x \geq 0 \land a > 0 \land b > 0 \vdash [v := b][x' = v]\, x \geq 0}[:=]}{x \geq 0 \land a > 0 \land b > 0 \vdash [v := a][x' = v]\, v \geq 0 \land [v := b][x' = v]\, x \geq 0}\land R}{x \geq 0 \land a > 0 \land b > 0 \vdash [v := a \cup v := b][\{x' = v\}]\, x \geq 0}[\cup]}{x \geq 0 \land a > 0 \land b > 0 \vdash [\big(v := a \cup v := b\big); \{x' = v\}]\, x \geq 0}[;]}{\vdash x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big(v := a \cup v := b\big); \{x' = v\}]\, x \geq 0}\rightarrow R$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; composeb(1) ; choiceb(1) ; andR(1) ; <(
  assignb(1) ; solve(1) ; QE,
  assignb(1) ; solve(1) ; QE
)
```

Rather than giving formula positions it is often more readable and more robust to change when explicitly indicating the formulas that the tactics are applied to in the following equivalent tactic:

```
implyR('R=="x>=0&a>0&b>0->[{v:=a;++v:=b;}{x'=v}]x>=0");
composeb('R=="[{v:=a;++v:=b;}{x'=v}]x>=0");
choiceb('R=="[v:=a;++v:=b;][{x'=v}]x>=0");
andR('R=="[v:=a;][{x'=v}]x>=0&[v:=b;][{x'=v}]x>=0"); <(
  "[v:=a;][{x'=v}]x>=0":
    assignb('R=="[v:=a;][{x'=v}]x>=0");
    solve('R=="[{x'=a}]x>=0");
    QE,
  "[v:=b;][{x'=v}]x>=0":
    assignb('R=="[v:=b;][{x'=v}]x>=0");
    solve('R=="[{x'=b}]x>=0");
    QE
)
```

Admittedly, these tactics are more complicated than the following automatic proof search, but it is also more insightful and gives the problem a more explicit structure:

```
auto
```

3.4 Loop Invariant Proofs

Loop invariants. This section assumes that you are familiar with Chapter 7: Loops & Invariants. If you have not read that chapter yet, simply skip this section and come back at another time.

Loop invariants are a fundamental part of every serious cyber-physical system and an inherent aspect of their design. The loop invariant J you identify is the crucial ingredient to prove box modalities of loops:

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

It is also easy to show that constant parameter assumptions from Γ, Δ can be kept around without any harm to the proof.

Example 3.9 (Loops in a fast car)**.** Let's add the loop back into the model from 3.8 where it belongs. We merely elided the loop first to have less things to worry about at once. Now let's conduct a proof for the original dL formula after adding the loop back where it belongs.

$$x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big((v := a \cup v := b);\ \{x' = v\}\big)^*]\, x \geq 0$$

First you are advised to identify the loop invariant yourself. It will be the most critical ingredient of the proof. Transliterated into KeYmaera X the loop verification question is:

```
x>=0 & a>0 & b>0 -> [{{v:=a; ++ v:=b;}; {x'=v}}*] x>=0
```

Load this example in KeYmaera X and after normalizing with rule →R use the loop rule and enter the loop invariant you found. Then complete the remaining three proof branches. If you use `x>=0` as the loop invariant, then the initial case and use case will immediately prove by id rule, because $x \geq 0$ is among the assumptions. The induction step case is almost identical to 3.8 and can be proved in the same way or with **Auto**. This is the resulting Bellerophon tactic, commented with labels for the individual proof branches:

```
implyR(1) ; loop("x>=0", 1) ; <(
  id,
  id,
  auto
)
```

It is usually more readable and robust to change when explicitly specifying the formulas that tactics are applied to and being explicit about the labels of the branches (`"Init":` for the initial condition, `"Post":` for the postcondition, and `"Step":` for the induction step):

```
implyR('R=="x>=0&a>0&b>0->[{{v:=a;++v:=b;}{x'=v}}*]x>=0");
loop("x>=0", 'R=="[{{v:=a;++v:=b;}{x'=v}}*]x>=0"); <(
  "Init": id,
  "Post": id,
  "Step": auto
)
```

3.5 @Invariant Annotation

Important (Loop @invariant annotation). Loop invariants are often more complicated than their postconditions, because other crucial information on the historical system behavior needs to be transported through the proof, as you have already seen in Chapter 7: Loops & Invariants. Thus, once you have found the loop invariant (see 3.4)), it is good practice to write it directly into the hybrid program. Not only will this make sure KeYmaera X does not need to ask you for it again and avoids expensive loop invariant search procedures, but it will also help you understand your system better in the future. That understanding of the loop invariant is particularly helpful when you change your system design with additional control cases. If you have written down the loop invariant, then you know what needs to be preserved when you change the controller. To record the loop invariant you found for your hybrid program, annotate it right after the repetition * with an `@invariant(J)` annotation that remembers the loop invariant J to use. For example, write something like:

```
{ctrl; plant}*@invariant(magicFormula)
```

Spaces are optional but sometimes make matters more readable.

Example 3.10 (Loop annotations in a fast car)**.** The loop invariant in 3.9 is extraordinarily easy, it is just the postcondition. But you should make it a habit to write down all loop invariants you found as part of the hybrid program. That speeds up proof search and makes your job easier, too, when you change the system design later.

```
x>=0 & a>0 & b>0 ->
[
```

```
  {
    {v:=a; ++ v:=b;}; {x'=v}
  }*@invariant(x>=0)
] x>=0
```

Load this example in KeYmaera X and use **Auto** to prove it. Admittedly, this worked just as well before the loop invariant annotation, but then it was using loop invariant search, which can take a lot of time or time out.

3.6 Optional: Chase-based Proofs

Chase proofs. Explicitly decomposing hybrid programs by explicitly specifying which axioms and proof rules are used to split it into its pieces has the advantage that it is exactly clear what happens. But it is also sometimes tedious and less robust to change of the model.

The alternative is to implicitly decompose hybrid programs by recursively decomposing them with the most systematic applicable axiom. This is what the `chaseAt` tactic does, which you can also invoke in KeYmaera X clicking **Option-left-click** or by **Alt-left-click** (unless your operating system swallows such commands in which case you need to type the tactic or change OS). The recursive decomposition by a chase chases a formula or program away until only its easier pieces are remaining or until a difficult decision needs to be reached in the proof. That is why it will stop where the action is: at loops or differential equations. Especially if you can already predict the outcome of such a recursive decomposition, chasing formulas away is a good idea.

Note that `chaseAt` will stop short of actually decomposing formulas in ways that would create new branches or sequent formulas. The **Unfold** button is similar and will also perform all canonical propositional or program decompositions top-down without branching and stop at loops or differential equations.

Example 3.11 (Chase a fast car)**.** Did you notice how the interactive proof in 3.8 was fully systematic? All it did was apply the most obvious axiom or proof rule at the most obvious position. That is a good sign about the systematics of dL proof construction and the basis for all its simple automation (although its advanced automation is significantly more sophisticated). But if you want to apply all the usual syntactic decomposition axioms without being specific about which ones they are, then the `chaseAt` tactic fits your purpose. Its intuition is that you point it at a formula and it will do all it can to chase it away without having to make any genuine commitments during proof search that you might regret. In fact, the `chaseAt` tactic is so useful that you can activate it anywhere in a sequent (also in context) just by clicking **Option-left-click** or **Alt-left-click** (unless your operating system swallows such commands in which case you need to type the tactic or change OS). But, by default, `chaseAt` is hesitant to branch the proof, so it will stop applying the canonical decompositions when it's time to branch or otherwise take a decision in the proof.

```
x>=0 & a>0 & b>0 -> [{v:=a; ++ v:=b;}; {x'=v}] x>=0
```

Load this example in KeYmaera X and use rule $\rightarrow$R followed by a chase with **Option-left-click** or **Alt-left-click** on the HP.

$$\dfrac{\dfrac{x \geq 0 \land a > 0 \land b > 0 \vdash [x' = a]\, x \geq 0 \land [x' = b]\, x \geq 0}{x \geq 0 \land a > 0 \land b > 0 \vdash [\big(v := a \cup v := b\big); \{x' = v\}]\, x \geq 0}\text{chase}}{\vdash x \geq 0 \land a > 0 \land b > 0 \rightarrow [\big(v := a \cup v := b\big); \{x' = v\}]\, x \geq 0}\rightarrow\text{R}$$

So what chaseAt has done in this case is to apply [;] followed by [∪] and then [:=] on both resulting conjuncts. From this point onwards it is easy to complete the proof by ∧R, solving the ODEs with **right-click** via axiom ['] and ℝ for quantifier elimination of the real arithmetic via **Tools->Real Arithmetic**. Here is the corresponding Bellerophon proof:

```
implyR(1) ; chaseAt(1) ; andR(1) ; <(
  solve(1) ; QE,
  solve(1) ; QE
  )
```

In addition to clearly communicating the intent of a proof succinctly, the above tactic is more robust when the structure of the system changes slightly, because chaseAt will still be the right tactic to use even if another assignment is added to the HP. For example, unlike the explicit proof tactic from 3.8, the exact same proof tactic works on the following dL formula instead:

```
x>=0 & a>0 & b>0 -> [{v:=a;v:=2*v; ++ v:=b;v:=v+1;}; {x'=v}] x>=0
```

This is an example where the above positional tactic is more general and more robust than an equivalent tactic that explicitly indicates the formulas rather than the formula positions:

```
implyR('R=="x>=0&a>0&b>0->[{v:=a;++v:=b;}{x'=v}]x>=0");
chaseAt('R=="[{v:=a;++v:=b;}{x'=v}]x>=0");
andR('R=="[{x'=a}]x>=0&[{x'=b}]x>=0"); <(
  "[{x'=a}]x>=0":
    solve('R=="[{x'=a}]x>=0");
    QE,
  "[{x'=b}]x>=0":
    solve('R=="[{x'=b}]x>=0");
    QE
)
```

Morally the same tactic ought to work on the modified problem, but of course the formulas are different and thus even the formula indicated in the first implyR tactic step cannot be found.

3.7 Optional: Reasoning in Context

Proofs in context. The chaseAt tactic that recursively decomposes a hybrid program or formula and chases it away can reason in context. But other axioms or reasoning principles can also be used to reason in the middle of a logical formula. This can be a clever technique to reduce the complexity of the verification effort, because it keeps reasoning steps in one place that belong together. It is also useful to conduct proofs that follow the order in which you would like to think about a problem.

Example 3.12 (Chase a fast car in context)**.** The chase-based proof of 3.11 was quite systematic. But it can be improved still. The chaseAt tactic can be used in context, rather than just at the top-level position. Notice how the ∧R rule really only makes sense for conjunctions that are at the top-level of a succedent? Otherwise ∧ might need a completely different proof treatment. But that's not the case for chaseAt which always makes sense no matter where. Consequently, there is no need to normalize the dL formula by →R rule

first. And, in fact, the ODE solution axiom ['] can also be used in any context, because it replaces one property of a differential equation with another equivalent dL formula with more quantifiers. Recall the fast car model

```
x>=0 & a>0 & b>0 -> [{v:=a; ++ v:=b;}; {x'=v}] x>=0
```

Load this example in KeYmaera X and **Option-left-click** or **Alt-left-click** to chase the program away by performing canonical recursive decompositions. Then **Click** on the differential equations to solve them and finally prove it with **Tools->Real Arithmetic**.

$$
\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash x\geq 0\land a>0\land b>0\rightarrow \forall t\,(t\geq 0\rightarrow x+at\geq 0)\land \forall t\,(t\geq 0\rightarrow x+bt\geq 0)}\mathbb{R}}{\vdash x\geq 0\land a>0\land b>0\rightarrow \forall t\,(t\geq 0\rightarrow x+at\geq 0)\land [x'=b]\,x\geq 0}[']}{\vdash x\geq 0\land a>0\land b>0\rightarrow [x'=a]\,x\geq 0\land [x'=b]\,x\geq 0}[']}{\vdash x\geq 0\land a>0\land b>0\rightarrow [(v:=a\cup v:=b);\{x'=v\}]\,x\geq 0}\text{chase}
$$

This chase-based proof in context corresponds to the following Bellerophon tactic:

```
chaseAt(1.1) ; solve(1.1.0) ; solve(1.1.1) ; QE
```

The periods in the positions indicate subexpressions. So first, the `chaseAt` tactic is applied to the second child of the first formula in the succedent, so the right-hand side of the implication. Sadly, subexpressions are indexed with base 0 while sequent formulas are indexed with base 1 or -1 respectively. Then the `solve` axiom is used at `1.1.0` so on the left conjunct of the right-hand side of the implication as well as at `1.1.1` so on the right conjunct of the right-hand side of the implication. Finally, real arithmetic quantifier elimination tactic `QE` completes the proof.

Why can `chaseAt` work so seamlessly in context rather than just at the top-level of a sequent? Well, because the axioms of dL are equivalences, and if A is equivalent to B then A is equivalent to B in any context, so A can be replaced by B, equivalently, even in the middle of any dL formulas or sequents. This is one indication for the deductive power and flexibility that stems immediately from dL's axiomatic approach.

The above tactic was short and sweet but reading it requires making sense of the subexpression indicators such as `1.1.0`. An equivalent tactic mentioning the particular formulas that are being used can be more readable. It is using the `#...#` notation to identify the appropriate subexpressions of the formula where the tactic is applied to:

```
chaseAt('R=="x>=0&a>0&b>0->#[{v:=a;++v:=b;}{x'=v}]x>=0#");
solve('R=="x>=0&a>0&b>0->#[{x'=a}]x>=0#&[{x'=b}]x>=0");
solve('R=="x>=0&a>0&b>0->\forall t_ (t_>=0->a*t_+x>=0)&#[{x'=b}]x>=0#");
QE
```

Example 3.13 (Reason with a fast car in context)**.** Reasoning in context is in no way limited to the use of `chaseAt` even if the `chaseAt` tactic excels at it. You can also follow your nose and do reasoning in context by applying axioms (and even some particularly flexible proof rules) where you see fit in the middle of logical formulas. Usually that would cause soundness concerns in prover implementations, but the flexible uniform substitution calculus of dL makes it quite seamless while protecting you from trying incorrect inferences. Recall the fast car model

```
x>=0 & a>0 & b>0 -> [{v:=a; ++ v:=b;}; {x'=v}] x>=0
```

Load this example in KeYmaera X and Now point to subformulas and **left-click** or **right-click** at your leisure to follow a proof without normalizing to sequent normal form first and finally prove it with **Tools->Real Arithmetic**. Try going inside out, for example.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\vdash x{\ge}0 \land a{>}0 \land b{>}0 \to \forall t\, (t \ge 0 \to x + at \ge 0) \land \forall t\, (t \ge 0 \to x + bt \ge 0)}\mathbb{R}}{\vdash x{\ge}0 \land a{>}0 \land b{>}0 \to [v := a]\forall t\, (t{\ge}0 \to x{+}vt{\ge}0) \land [v := b]\forall t\, (t{\ge}0 \to x{+}vt{\ge}0)}[:=]}{\vdash x{\ge}0 \land a{>}0 \land b{>}0 \to [v := a \cup v := b]\forall t\, (t \ge 0 \to x + vt \ge 0)}[\cup]}{\vdash x{\ge}0 \land a{>}0 \land b{>}0 \to [v := a \cup v := b][x' = v]\, x \ge 0}[']}{\vdash x{\ge}0 \land a{>}0 \land b{>}0 \to [(v := a \cup v := b); \{x' = v\}]\, x \ge 0}[;]
$$

Notice that you only have to handle the differential equation once in this proof, because it was solved before splitting off the two branches of the nondeterministic choice $v := a \cup v := b$ into separate conjuncts. Here is the corresponding Bellerophon proof tactic:

```
composeb(1.1); solve(1.1.1); choiceb(1.1); assignb(1.1.0); assignb(1.1.1); QE
```

Here is the same tactic explicitly indicating the formulas rather than their positions:

```
composeb('R=="x>=0&a>0&b>0->#[{v:=a;++v:=b;}{x'=v}]x>=0#");
solve('R=="x>=0&a>0&b>0->[v:=a;++v:=b;]#[{x'=v}]x>=0#");
choiceb('R=="x>=0&a>0&b>0->#[v:=a;++v:=b;]\forall t_ (t_>=0->v*t_+x>=0)#");
assignb('R=="x>=0&a>0&b>0->#[v:=a;]\forall t_ (t_>=0->v*t_+x>=0)#&[v:=b;]\forall t_
assignb('R=="x>=0&a>0&b>0->\forall t_ (t_>=0->a*t_+x>=0)&#[v:=b;]\forall t_ (t_>=0->v
QE
```

Reasoning in context can often help reduce redundancies by keeping common reasoning together and cutting down on the number of times that proof steps need to be performed across branches. Reasoning in contact by simply pointing to where you want to continue the proof is also an effective technique to make sure your proof follows your intuition about the system. KeYmaera X prevents you from accidentally making incorrect inferences even when reasoning right in the middle of a logical formula.

3.8 Optional: Monotone Generalization

Monotone generalization. Working with generalizations in the middle of the proof can be an effective way of decomposing the proof into even fewer modular pieces. Clever uses of generalizations are an effective means of reducing the complexity of a verification effort. Generalizations with the monotonicity rules proves a postcondition P of a modality by instead proving a more general postcondition Q that implies P:

$$
\text{MR}\ \frac{\Gamma \vdash [\alpha]Q, \Delta \quad Q \vdash P}{\Gamma \vdash [\alpha]P, \Delta} \qquad \text{MR}\ \frac{\Gamma \vdash \langle\alpha\rangle Q, \Delta \quad Q \vdash P}{\Gamma \vdash \langle\alpha\rangle P, \Delta}
$$

Example 3.14 (Fast car with monotone generalization)**.** Reasoning in context made it possible to prove 3.12 with only a single use of the solution axiom. That saved effort but required reasoning in context. Another technique is to work with generalizations that can also help find considerable shortcuts in proofs.

```
x>=0 & a>0 & b>0 -> [{v:=a; ++ v:=b;}; {x'=v}] x>=0
```

Load this example in KeYmaera X and use rule →R followed by axiom [;] with a **left-click** or **right-click** on the HP. Then **right-click** $[v := a \cup v := b] \dots$ and select monotonicity rule MR entering the intermediate condition `x>=0&v>=0` for Q, which summarize all knowledge that is important to retain after the controller ran to ensure the ODE remains safe. On the left branch **left-click** or **right-click** to use axiom [∪] on $[v := a \cup v := b](x \geq 0 \land v \geq 0)$, then use axiom [:=] on $v := a$ and on $v := b$ and eliminate quantifiers with $\mathbb{R}$ in the resulting real arithmetic via **Tools->Real Arithmetic**. On the right branch **left-click** or **right-click** to solve the ODE of $[\{x' = v\}]\, x \geq 0$ with axiom [′]. then eliminate quantifiers with $\mathbb{R}$ in the resulting real arithmetic via **Tools->Real Arithmetic**. This results in the following proof:

$$
\dfrac{\dfrac{\dfrac{\dfrac{*}{x{\geq}0{\land}a{>}0{\land}b{>}0 \vdash (x{\geq}0 \land a{\geq}0) \land (x{\geq}0 \land b{\geq}0)}\mathbb{R}}{x{\geq}0{\land}a{>}0{\land}b{>}0 \vdash [v := a](x{\geq}0{\land}v{\geq}0){\land}[v := b](x{\geq}0{\land}v{\geq}0)}[:=]}{x{\geq}0{\land}a{>}0{\land}b{>}0 \vdash [v := a \cup v := b](x \geq 0 \land v \geq 0)}[\cup] \quad \dfrac{\dfrac{\dfrac{*}{x{\geq}0{\land}v{\geq}0 \vdash \forall t{\geq}0\, x + vt \geq 0}\mathbb{R}}{x{\geq}0{\land}v{\geq}0 \vdash \forall t{\geq}0\, [x := x + vt]x \geq 0}[:=]}{x{\geq}0{\land}v{\geq}0 \vdash [\{x' = v\}]\, x \geq 0}['] }{\dfrac{x \geq 0 \land a > 0 \land b > 0 \vdash [v := a \cup v := b][\{x' = v\}]\, x \geq 0}{\dfrac{x \geq 0 \land a > 0 \land b > 0 \vdash [(v := a \cup v := b); \{x' = v\}]\, x \geq 0}{\vdash x{\geq}0{\land}a{>}0{\land}b{>}0{\to}[(v := a \cup v := b); \{x' = v\}]x{\geq}0}\to\text{R}}[;]}\text{MR}
$$

Here is the corresponding Bellerophon proof:

```
implyR(1) ; composeb(1) ; MR("x>=0&v>=0", 1) ; <(
  choiceb(1) ; assignb(1.0) ; assignb(1.1) ; QE,
  solve(1) ; QE
  )
```

This proof style of working with generalizations via the monotonicity rule MR can be very helpful to decompose the proof into smaller pieces and carry forward only what is necessary for the correctness of the system.

Indeed, KeYmaera X is clever enough to retain assumptions about variables that do not change, so the intermediate condition `v>=0` would have been sufficient, because x does not change during the discrete controller yet. This would have given the following proof

$$
\dfrac{\dfrac{\dfrac{\dfrac{*}{x{\geq}0 \land a{>}0 \land b{>}0 \vdash a{\geq}0 \land b{\geq}0}\mathbb{R}}{x{\geq}0 \land a{>}0 \land b{>}0 \vdash [v := a]v{\geq}0 \land [v := b]v{\geq}0}[:=]}{x{\geq}0 \land a{>}0 \land b{>}0 \vdash [v := a \cup v := b]v \geq 0}[\cup] \quad \dfrac{\dfrac{\dfrac{*}{x \geq 0 \land v \geq 0 \vdash \forall t{\geq}0\, x + vt \geq 0}\mathbb{R}}{x \geq 0 \land v \geq 0 \vdash \forall t{\geq}0\, [x := x + vt]x \geq 0}[:=]}{x \geq 0 \land v \geq 0 \vdash [\{x' = v\}]\, x \geq 0}[']}{\dfrac{x \geq 0 \land a > 0 \land b > 0 \vdash [v := a \cup v := b][\{x' = v\}]\, x \geq 0}{\dfrac{x \geq 0 \land a > 0 \land b > 0 \vdash [(v := a \cup v := b); \{x' = v\}]\, x \geq 0}{\vdash x{\geq}0{\land}a{>}0{\land}b{>}0 \to [(v := a \cup v := b); \{x' = v\}]\, x \geq 0}\to\text{R}}[;]}\text{MR}
$$

Here is the corresponding Bellerophon proof:

```
implyR(1) ; composeb(1) ; MR("v>=0", 1) ; <(
  choiceb(1) ; assignb(1.0) ; assignb(1.1) ; QE,
  solve(1) ; QE
  )
```

Here is the same tactic explicitly indicating the formulas to which the tactics are applied instead of their positions, whose labels makes it clearer what part of the proof goes where:

```
implyR('R=="x>=0&a>0&b>0->[{v:=a;++v:=b;}{x'=v}]x>=0");
composeb('R=="[{v:=a;++v:=b;}{x'=v}]x>=0");
MR("v>=0", 'R=="[v:=a;++v:=b;][{x'=v}]x>=0"); <(
  "Use Q->P":
    choiceb('R=="[v:=a;++v:=b;]v>=0");
    assignb('R=="#[v:=a;]v>=0#&[v:=b;]v>=0");
    assignb('R=="a>=0&#[v:=b;]v>=0#");
    QE,
  "Show [a]Q":
    solve('R=="[{x'=v}]x>=0");
    QE
)
```

3.9 Advanced: Loop Variant Proofs

Loop variants by convergence. This section assumes that you have read about the convergence rule in section 17.4 on Repetitive Diamonds Convergence Versus Iteration of Chapter 17: Game Proofs & Separations, which can be read without first understanding games. If you have not read that section yet, simply come back at another time.

Just like loop *in*variants, loop *variant*s are a fundamental part of every serious cyber-physical system and an inherent aspect of their design. The difference is that loop *in*variants enable to you show their safety by identifying what never changes and is always true while the system runs. Loop variants, by contrast, identify what changes in a system ans is ultimately true when the system runs long enough in a suitable way. Loop invariants help us understand what we can rely on no matter how long the CPS evolved, while loop variants explain what the CPSs achieve and where they eventually get to. A loop variant $p(v)$ you identify is the crucial ingredient to prove diamond modalities of loops:

$$\text{con}\ \frac{\Gamma \vdash \exists v\, p(v), \Delta \quad \exists v{\le}0\, p(v) \vdash Q \quad \vdash \forall v{>}0\, (p(v) \rightarrow \langle\alpha\rangle p(v-1))}{\Gamma \vdash \langle\alpha^*\rangle Q, \Delta} \quad (v \notin \alpha)$$

Loop variants can perfectly include loop invariants, which are also ultimately achieved since those loop invariant parts even always remain true after running α. Their more exciting part, however, expresses what progress α can make when run in a suitable way so that it gets closer toward the goal Q . The premise for the initial case requires that there is a distance, v, toward the goal for which the abstract progress measure $p(v)$ holds. The premise for the use case requires that if the distance $v \le 0$ for which the abstract progress measure $p(v)$ holds has become nonpositive, then you are at the goal Q. And the premise for the induction step requires that while not at the goal yet, so for all positive distances $v > 0$ for which the abstract progress measure $p(v)$ holds, there is a way of running α such that one is one round closer so $p(v-1)$ holds. To record the loop variant you found for your hybrid program, annotate it right after the repetition `*` with an `@variant(v,p(v))` annotation that remembers the loop variant $p(v)$ to use for $p(v)$ along with the fresh variable v that is used to measure progress. For example, write something like:

```
{ctrl; plant}*@variant(v,magicFormula)
```

Spaces are optional but sometimes make matters more readable.

Example 3.15 (Fast cars get to the goal)**.** Example 3.9 made a good case, meaning a rigorous dL proof, why fast cars are safe in the (very limited) sense of always keeping a nonnegative position. That's good to know and all, but fast cars really desperately need more properties proved. One thing would be a serious safety property such as collision freedom, which would be a good exercise you the interested reader to pursue. But another thing is that you probably want your fast car to be, well, fast. Or rather, what you expect a fast car to be able to do is to ultimately make it all the way to position 11 even if it merely may have started at position 0. Then the desirable $x \geq 11$ most certainly will not always be true after the HP of your fast car model. It will only eventually be true if you wait long enough and, maybe, take the appropriate control decisions. This is where the diamond modality of dL comes in handy to say that there is a run of the fast car model to a state where $x \geq 11$:

$$x \geq 0 \land a > 0 \land b > 0 \to \langle \big((v := a \cup v := b);\ x' = v \big)^* \rangle\, x \geq 11$$

The key ingredient for its proof is the identification of its loop variant $p(v)$. Can you find it out before you read on? Here is the question transliterated to KeYmaera X:

```
x>=0 & a>0 & b>0 -> <{{v:=a; ++ v:=b;}; {x'=v}}*> x>=11
```

Load this example in KeYmaera X and after using →R handle the remaining diamond property of the loop with rule con and specify convergence variable `v` with loop variant `x*a>=11*a-n` (which is equivalent to `x>=11-n/a` but does not need $a > 0$ to be meaningful). Finally finish the proof as usual using $\langle:=\rangle$ and $\langle'\rangle$ before $\mathbb{R}$ via **Tools->Real Arithmetic**. Here is the dL proof that, for space reasons, is assuming $a > 0 \land b > 0$ are fixed numbers inserted everywhere:

$$\dfrac{\dfrac{\dfrac{*}{x{\geq}0 \vdash \exists n\, xa \geq 11a - n}\,\mathbb{R} \qquad \dfrac{*}{\exists n{\leq}0\, xa \geq 11a - n \vdash x \geq 11}\,\mathbb{R} \qquad ①}{x \geq 0 \vdash \langle \big((v := a \cup v := b); x' = v \big)^* \rangle\, x \geq 11}\,\text{con}}{\vdash x \geq 0 \to \langle \big((v := a \cup v := b); x' = v \big)^* \rangle\, x \geq 11}\,{\to}\text{R}$$

The proof of the third premise, marked ① above proves as follows:

$$*$$
$$\mathbb{R}\quad n{>}0, xa \geq 11a - n \vdash \exists t{\geq}0\, (x + at)a \geq 11a - (n-1)$$
$$\langle'\rangle\quad n{>}0, xa \geq 11a - n \vdash \langle x' = a \rangle xa \geq 11a - (n-1)$$
$$\langle:=\rangle\quad n{>}0, xa \geq 11a - n \vdash \langle v := a \rangle \langle x' = v \rangle xa \geq 11a - (n-1)$$
$$\text{WR}\quad n{>}0, xa \geq 11a - n \vdash \langle v := a \rangle \langle x' = v \rangle xa \geq 11a - (n-1), \langle v := b \rangle \langle x' = v \rangle xa \geq 11a - (n-1)$$
$$\lor\text{R}\quad n{>}0, xa \geq 11a - n \vdash \langle v := a \rangle \langle x' = v \rangle xa \geq 11a - (n-1) \lor \langle v := b \rangle \langle x' = v \rangle xa \geq 11a - (n-1)$$
$$\langle\cup\rangle\quad n{>}0, xa \geq 11a - n \vdash \langle v := a \cup v := b \rangle \langle x' = v \rangle xa \geq 11a - (n-1)$$
$$\langle;\rangle\quad n{>}0, xa \geq 11a - n \vdash \langle \big(v := a \cup v := b\big); x' = v \rangle xa \geq 11a - (n-1)$$
$$\to\text{R}\quad \vdash n{>}0 \to (xa \geq 11a - n \to \langle \big(v := a \cup v := b\big); x' = v \rangle xa \geq 11a - (n-1))$$
$$\forall\text{R}\quad \vdash \forall n{>}0\, (xa \geq 11a - n \to \langle \big(v := a \cup v := b\big); x' = v \rangle xa \geq 11a - (n-1))$$

Here is the corresponding Bellerophon tactic:

```
implyR(1) ; con("n", "x*a>=11*a-n", 1) ; <(
  QE,
  QE,
  composed(1); choiced(1); orR(1); hideR(2=="<v:=b;><{x'=v}>x*a>=11*a-(n-1)");
```

```
  assignd(1); solve(1); QE
)
```

An equivalent tactic that is explicit about the formulas that its tactic parts are being applied to is the following:

```
implyR('R=="x>=0&a>0&b>0-><{{v:=a;++v:=b;}{x'=v}}*>x>=11");
con("n", "x*a>=11*a-n", 'R=="<{{v:=a;++v:=b;}{x'=v}}*>x>=11"); <(
  "Init": QE,
  "Post": QE,
  "Step":
    composed('R=="<{v:=a;++v:=b;}{x'=v}>x*a>=11*a-(n-1)");
    choiced('R=="<v:=a;++v:=b;><{x'=v}>x*a>=11*a-(n-1)");
    orR('R=="<v:=a;><{x'=v}>x*a>=11*a-(n-1)|<v:=b;><{x'=v}>x*a>=11*a-(n-1)");
    hideR('R=="<v:=b;><{x'=v}>x*a>=11*a-(n-1)");
    assignd('R=="<v:=a;><{x'=v}>x*a>=11*a-(n-1)");
    solve('R=="<{x'=a}>x*a>=11*a-(n-1)");
    QE
)
```

Example 3.16 (Loop annotations get fast cars to the goal)**.** The loop variant in 3.15 was not that hard to found. But you should make it a habit to write down all loop variants as part of the hybrid program. That speeds up proof search and makes your job easier, too, when you change the system design later.

```
x>=0 & a>0 & b>0 ->
<
  {
    {v:=a; ++ v:=b;}; {x'=v}
  }*@variant(n, x*a>=11*a-n)
> x>=11
```

Load this example in KeYmaera X and use **Auto** to prove it or manually conduct a proof.

Example 3.17 (Loop annotations get fast cars positively to the goal with variant and invariant)**.** A common part of loop variants are loop invariants that identify what does *not* change during the hybrid program because they do not depend on the variance variable `n`.

```
x>=0 & a>0 & b>0 ->
<
  {
    {v:=a; ++ v:=b;}; {x'=v}
  }*@variant(n, x*a>=11*a-n & 0<=x)
> (x>=11 & 0<=x)
```

Load this example in KeYmaera X and use **Auto** to prove it or manually conduct a proof. Loop invariants can simply be written down as part of the `@variant` annotations. While this particular loop invariant, `0<=x`, is not exciting when `x>=11` is already shown, the same idea applies for more exciting hybrid programs.

For example, the following car gets faster on every choice of `v:=a`, so `a>0` is a loop invariant, but $xa \geq 11a - n$ is still the crucial loop variant:

```
x>=0 & a>0 & b>0 ->
<
  {
    {v:=a; a:=a+1; ++ v:=b;}; {x'=v}
  }*@variant(n, x*a>=11*a-n & a>0)
> x>=11
```

Load this example in KeYmaera X and use **Auto** to prove it or manually conduct a proof.

A much more subtle example is the following, in which `a>0` is indeed not actually an invariant of the corresponding loop with a box modality, because the right choice `v:=b` would decrease `a`. Nevertheless, the conjunction $xa \geq 11a - n \land a > 0$ of a progress measure and a formula independent of n is a provable loop variant, because the appropriate choice of the loop body's execution in the diamond is the choice `v:=a`, in which case $a > 0$ is preserved:

```
x>=0 & a>0 & b>0 ->
<
  {
    {v:=a; a:=a+1; ++ v:=b; a:=a-1;}; {x'=v}
  }*@variant(n, x*a>=11*a-n & a>0)
> x>=11
```

Load this example in KeYmaera X and use **Auto** to prove it or manually conduct a proof.

3.10 Advanced: Loop Fixpoint Proofs

Loop fixpoint. This section assumes that you have read about the fixpoint rule in section 16.3.10 on Proof Rules for Repetition Games of Chapter 16: Winning & Proving Hybrid Games, which you can also venture to read without first understanding games. If you have not read that section yet, simply come back at another time.

Just like loop invariants, and loop variants, *fixpoints* enable you to prove properties of loops. While loop fixpoints are not dissimilar to loop invariants, they have a fairly different intuition and are best used for making use of assumptions of the form $\langle\alpha^*\rangle P$. That is, whenever one of your assumptions says that a loop can be repeated to reach a state satisfying a certain condition, then you are able to take advantage of that assumption to read off further assumptions. While the loop invariant and loop variant proof rules apply to properties of loops to be shown, because they occur in the succedent, the fixpoint rule applies to diamond properties of loops that can be assumed, because they occur in the antecedent.

$$\text{fp}\ \frac{\Gamma, \langle\alpha^*\rangle P, J \vdash \Delta \quad P \lor \langle\alpha\rangle J \vdash J}{\Gamma, \langle\alpha^*\rangle P \vdash \Delta}$$

Soundness of the fixpoint rule fp directly follows from the fact that the semantics of loops is the least fixpoint (which works perfectly in both hybrid systems and hybrid games). The second premise of the fixpoint rule fp shows that the formula J also is a prefixpoint of α: $P \lor \langle\alpha\rangle J \vdash J$. That is, if either the original postcondition P is true or if there is a way of following α to a state where J holds true, then the fixpoint formula J is already true in

the initial state. Since $\langle\alpha^*\rangle P$, however, is the *least* fixpoint, J follows from $\langle\alpha^*\rangle P$, so J can safely be assumed in the first premise since $\langle\alpha^*\rangle P$ was already assumed in the antecedent of the conclusion. The trick with using the fixpoint rule is to identify a prefixpoint of α from whose truth you can learn something of interest to prove Δ.

Example 3.18 (Even cars have fixpoints)**.** Let's assume that there is a way of repeating a car control loop to a negative position and see what we can read off from that.

$$\langle\big(\big(v := a \cup v := b\big);\ \{x' = v\}\big)^*\rangle\, x < 0$$

If this formula is true, then we either already start at a negative position or either the a or the b velocity that can be chosen must be negative, so waiting long enough and repeating often enough will make x negative. Otherwise there just is no way of making x negative ever. That is, we conjecture, say, the following dL formula is true:

$$\langle\big(\big(v := a \cup v := b\big);\ \{x' = v\}\big)^*\rangle\, x < 0 \rightarrow x < 1 \lor a < 0 \lor b < 0$$

Conducting a proof in dL is best done using the fixpoint rule fp (although there are other ways). First you are advised to identify the loop fixpoint condition to use for the proof yourself. It will be the most critical ingredient of the proof. Transliterated into KeYmaera X the question is:

```
<{{v:=a; ++ v:=b;}; {x'=v}}*>x<0 -> x<1 | a<0 | b<0
```

Load this example in KeYmaera X and after normalizing with rule →R use the fp rule and enter the loop fixpoint formula J you found. Then complete the remaining two proof branches. If you simply use the postcondition `x<1 | a<0 | b<0` as the loop fixpoint formula J, then the first premise will immediately prove by id rule, because `x<1 | a<0 | b<0` is among the assumptions. The rest of the proof will also succeed easily after decomposing the hybrid program and solving the ODE and QE.

Here is the dL proof that just writes J in place of the fixpoint formula `x<1 | a<0 | b<0`:

$$
\dfrac{\dfrac{\overset{*}{}}{x{<}1{\lor}a{<}0{\lor}b{<}0 \vdash x{<}1{\lor}a{<}0{\lor}b{<}0}\,\text{id}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{*}{}}{x < 0 \lor \exists t{\ge}0\, \langle x := x + va\rangle J \lor \exists t{\ge}0\, \langle x := x + vb\rangle J \vdash}\,\langle{:=}\rangle,\mathbb{R}}{x < 0 \lor \langle v := a\rangle \exists t{\ge}0\, \langle x := x + vt\rangle J \lor \langle v := b\rangle \exists t{\ge}0\, \langle x := x + vt\rangle J \vdash}\,\langle{:=}\rangle}{x < 0 \lor \langle v := a \cup v := b\rangle \exists t{\ge}0\, \langle x := x + vt\rangle J \vdash}\,\langle\cup\rangle}{x < 0 \lor \langle v := a \cup v := b\rangle \langle x' = v\rangle J \vdash}\,\langle{'}\rangle}{x < 0 \lor \langle (v := a \cup v := b); \{x' = v\}\rangle J \vdash}\,\langle{;}\rangle}
{\dfrac{\langle\big(\big(v := a \cup v := b\big);\ \{x' = v\}\big)^*\rangle\, x < 0 \vdash x < 1 \lor a < 0 \lor b < 0}{\vdash \langle\big(\big(v := a \cup v := b\big);\ \{x' = v\}\big)^*\rangle x{<}0 \rightarrow x{<}1{\lor}a{<}0{\lor}b{<}0}\,\rightarrow\text{R}}\,\text{fp}
$$

Here is a similar Bellerophon tactic that additionally uses ∨L to split the proof on the right premise:

```
implyR(1);
fp("x < 1|a < 0|b < 0", -1); <(
  "Usefix":
    id,
  "Fixpoint":
```

```
    orL(-1); <(
        QE,
        composed(-1);
        solve(-1.1);
        choiced(-1);
        assignd(-1.0);
        assignd(-1.1);
        QE
    )
)
```

An equivalent tactic that is explicit about the formulas that its tactic parts are being applied to is the following:

```
implyR('R=="<{{v:=a;++v:=b;}{x'=v}}*>x<=0->x < 1|a < 0|b < 0");
fp("x < 1|a < 0|b < 0", 'L=="<{{v:=a;++v:=b;}{x'=v}}*>x<=0"); <(
  "Usefix":
    id,
  "Fixpoint":
    orL('L=="x<=0|<{v:=a;++v:=b;}{x'=v}>(x < 1|a < 0|b < 0)"); <(
      "x<=0":
        QE,
      "<{v:=a;++v:=b;}{x'=v}>(x < 1|a < 0|b < 0)":
        composed('L=="<{v:=a;++v:=b;}{x'=v}>(x < 1|a < 0|b < 0)");
        solve('L=="<v:=a;++v:=b;>#<{x'=v}>(x < 1|a < 0|b < 0)#");
        choiced('L=="<v:=a;++v:=b;>\exists t_ (t_>=0&(v*t_+x < 1|a < 0|b < 0))");
        assignd('L=="#<v:=a;>\exists t_ (t_>=0&(v*t_+x < 1|a < 0|b < 0))#|<v:=b;>\ex
        assignd('L=="\exists t_ (t_>=0&(a*t_+x < 1|a < 0|b < 0))|#<v:=b;>\exists t_
        QE
    )
)
```

Alternatively we could have used a slightly different fixpoint formula such as `x<0 | a<0 | b<0` to succeed with the proof, because there is a fair amount of choice among the fixpoint formulas.

Example 3.19 (Game have fixpoints too)**.** The fixpoint rule works just as well for hybrid games. For example the following conjecture proves easily using `x<=1` as a fixpoint formula J:

```
<{x:=0;--x:=x+1;}*>x<=0 -> x<=1
```

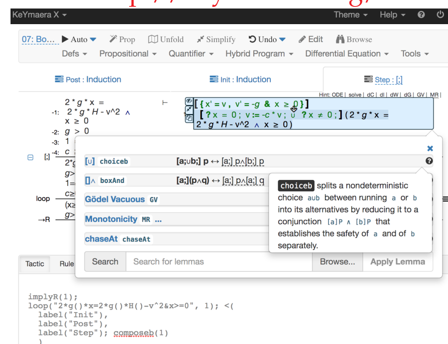
Chapter 4

Differential Equation Proofs in KeYmaera X

Welcome to the KeYmaera X Tutorial in which you will learn how to use the KeYmaera X aXiomatic Tactical Theorem Prover for Hybrid Systems from a pragmatic perspective.

KeYmaera X KeYmaera X is a theorem prover for *differential dynamic logic* (dL), a logic for specifying and verifying properties of *hybrid systems* with mixed discrete and continuous dynamics. This tutorial provides practical tool aspects and is complementary to the textbook *Logical Foundations of Cyber-Physical Systems*, in which provides comprehensive information on differential dynamic logic can be found. KeYmaera X is available at

http://keymaeraX.org/



Part Summary This part will explore how differential equation proofs in differential dynamic logic can be conducted in KeYmaera X with an emphasis on complicated differential equations that cannot be replaced by their solution. Proving properties of differential equations is one of the fundamental challenges in advanced cyber-physical systems and supported with an array of techniques in KeYmaera X.

So far, this tutorial focused on verifying safety and liveness of hybrid systems with fairly simple differential equations. Simple enough differential equations can be replaced by their solutions during the verification as long as the resulting arithmetic remains decid-

able. That is the case with a small class of linear differential equations (nilpotent ODEs) whose solutions are polynomial functions.

Now, this part of the tutorial considers more interesting differential equations for advanced cyber-physical systems. KeYmaera X provides significant proof automation including full logical decision procedures for differential equation invariants. These differential equation invariance proofs are built out of modular differential equation reasoning principles that have compelling intuitions. *Differential invariants* enable local reasoning about local change of truth in differential form. *Differential cuts* accumulate knowledge about the evolution of an ODE from multiple proofs. *Differential ghosts* add differential equations for new ghost variables to the existing system of differential equations enabling reasoning about the historical evolution of ODE systems in integral form. Understanding these techniques makes it possible to understand why differential equation invariants are true. The ability to use these techniques also helps generate invariants and makes it possible to conduct faster proofs and scale verification to more complicated cyber-physical systems. Before you face systems that are so complicated that automatic proofs become stuck, however, you should practice with proofs on simpler examples to understand how differential equation proofs work.

Background This tutorial assumes that you have read or refer to the following chapters in the *Logical Foundations of Cyber-Physical Systems* textbook for background information on the principles as needed:

- Chapter 10: Differential Equations & Differential Invariants
- Chapter 11: Differential Equations & Proofs
- Chapter 12: Ghosts & Differential Ghosts

4.1 Solving Differential Equations

Differential equation solving. The conceptually easiest way to handle differential equations in a proof is to solve them. That is, to replace a property of the ODE with a property of its solution quantified over time $t \geq 0$:

$$['] \; [x' = f(x)]p(x) \leftrightarrow \forall t{\geq}0\, [x := x(t)]p(x) \quad (x'(t) = f(x))$$

Of course, it is crucial for the use of this axiom $[']$ to prove that $x(t)$ solves the differential equation $x'(t) = f(x)$ and has the symbolic initial value $x(0) = x$ and that t is fresh. The advantage of the differential equation solution axiom $[']$ is that it is conceptually straightforward, that it works for any postcondition $p(x)$, and that it is an equivalence that can be used in any context. The most obvious downside is that only a very limited number of differential equations have closed-form solutions that are simple enough to handle the resulting quantified arithmetic. Nevertheless, solutions are one canonical way of understanding differential equations, so we consider solutions first before moving on to techniques that work that need no solutions.

Example 4.1 (Fast car)**.** Recall the fast car from 3.8, which we now simplify a little by taking away one of its gears just to save some space. A car with only a fast gear will always have a nonnegative velocity:

$$x \geq 0 \land a > 0 \rightarrow [v := a;\ \{x' = v\}]\, x \geq 0$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X, without a loop:

```
x>=0 & a>0 -> [v:=a; {x'=v}] x>=0
```

Load this example in KeYmaera X and don't click **Auto** because that proves so fast that you hardly see what's going on. Instead Use $\rightarrow$R to move the assumption to the left-hand side of the turnstile and then split the sequential composition using the axiom [;] by either a **left-click** or a **right-click** selecting axiom [;]. Then use axiom [:=] to substitute in the assignment into the differential equation and solve it with **right-click** via axiom ['] using the solution $x(t) = x + at$. Then eliminate the quantifiers in the resulting arithmetic by $\mathbb{R}$ via **Tools->Real Arithmetic**. This gives you the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x \geq 0 \wedge a > 0 \vdash \forall t\,(t \geq 0 \rightarrow x + at \geq 0)}\mathbb{R}}{x \geq 0 \wedge a > 0 \vdash [x' = a]\, x \geq 0}[']}{x \geq 0 \wedge a > 0 \vdash [v := a][x' = v]\, x \geq 0}[:=]}{x \geq 0 \wedge a > 0 \vdash [v := a; \{x' = v\}]\, x \geq 0}[;]}{\vdash x \geq 0 \wedge a > 0 \rightarrow [v := a; \{x' = v\}]\, x \geq 0}\rightarrow\!R$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; composeb(1) ; assignb(1) ; solve(1) ; QE
```

The only downside of the `solve` tactic corresponding to axiom ['] is that only a very limited number of differential equations have closed-form solutions in decidable arithmetic.

4.2 Invariants for Differential Equations

Invariants for ODEs. The `ode` tactic bundles up sophisticated logical algorithms for automatically producing proofs of properties of differential equations. For learning how to use KeYmaera X and help it scale beyond the reach of fully automatic verification, however, it is best not to use it until you understand the individual techniques that are discussed further below in this tutorial part. Your system insights about your CPS always enable you to do more clever proofs than any automation would be capable of.

Example 4.2 (Rotational dynamics)**.** Consider a differential equation characterizing the rotation of the point v, w on a circle of radius r around the origin:

$$v^2 + w^2 = r^2 \rightarrow [v' = w, w' = -v]\, v^2 + w^2 = r^2$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
v^2+w^2=r^2 -> [{v'=w,w'=-v}] v^2+w^=r^2
```

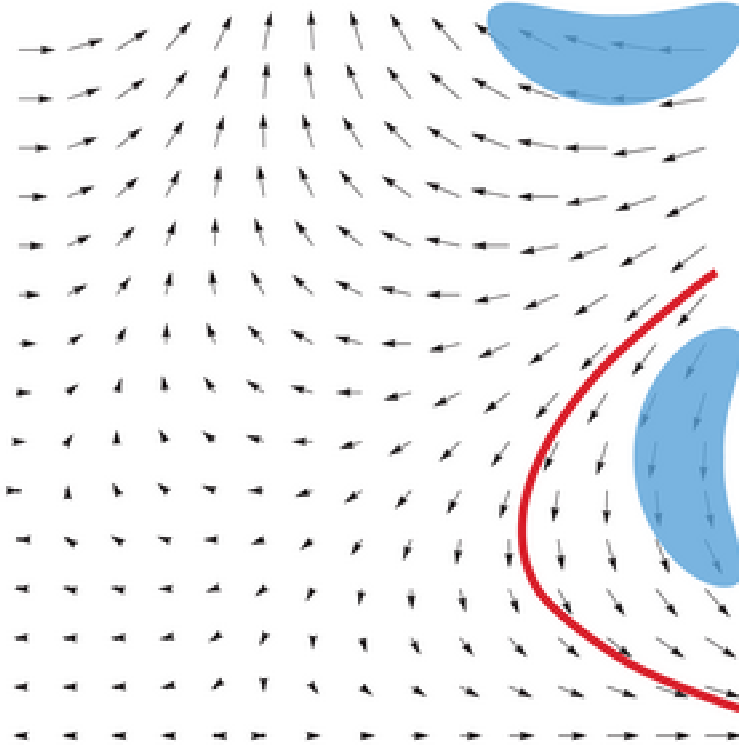
Load this example in KeYmaera X and **left-click** to use $\rightarrow$R to move the assumption into the antecedent. Then if you try to solve the ODE by ['], it will report an error, because the solution is not polynomial (but a more complicated mix of scaled trigonometric functions). Unlike polynomials or rational functions, trigonometric functions would give rise to undecidable arithmetic. Instead, **left-click** to select the automatic `ODE` tactic. In this case, the exact same proof would have been conducted if you simply click **Auto** right away. But neither proof was particularly illuminating because almost all proof steps were done automatically. Let's investigate more insightful proofs next.

4.3 Differential Invariants

Differential invariants. The solutions axiom ['] is easy to understand and works as a genuine equivalence, but its downside is that it is only applicable for the few differential equation systems sufficiently simple solutions (polynomials or rational functions).

The differential invariance rule dI works for significantly more general differential equations. It proves a postcondition F of an ODE by proving its differential $(F)'$, which investigates the rate of change of the truth of F. Intuitively, if F starts out true and is only getting "more" true along $x' = f(x)$ then it remains true always. KeYmaera X implements the rule dI in the following form by explicitly proving the postcondition F in the first premise and proving the differential $(F)'$ after the assignment $[x':=f(x)]$ and assuming the evolution domain constraint Q in the its premises:

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



It is also easy to show that constant parameter assumptions from Γ, Δ can be kept around without any harm to the proof.

Example 4.3 (Differential invariants for rotational dynamics)**.** Consider a minor variation of the dynamics from 4.2 for the rotation of the point v, w on a circle of *at most* radius r around the origin:

$$v^2 + w^2 \leq r^2 \to [v' = w, w' = -v]\, v^2 + w^2 \leq r^2$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
v^2+w^2<=r^2 -> [{v'=w,w'=-v}] v^2+w^2<=r^2
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential invariance rule dI. The initial case in the left premise proves by id via **Propositional->Identity**. The differential case in the right premise proves after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with ℝ via **Tools->Real Arithmetic**. This gives the following proof:

$$\dfrac{\dfrac{\dfrac{*}{v^2{+}w^2{\le}r^2 \vdash v^2 + w^2 \le r^2}\,\text{id} \qquad \dfrac{\dfrac{\dfrac{*}{\vdash 2vw + 2w(-v) \le 0}\,\mathbb{R}}{\vdash [w'{:=}-v]\, 2vw + 2ww' \le 0}\,[:=]}{\vdash [v'{:=}w][w'{:=}-v]\, 2vv' + 2ww' \le 0}\,[:=]}{v^2 + w^2 \le r^2 \vdash [v' = w, w' = -v]\, v^2 + w^2 \le r^2}\,\text{dI}}{\vdash v^2 + w^2 \le r^2 \to [v' = w, w' = -v]\, v^2 + w^2 \le r^2}\,\to\text{R}$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; dIRule(1) ; <(
  id,
  Dassignb(1) ; Dassignb(1) ; QE
  )
```

The following equivalent tactic that explicitly indicates the formulas and branch labels:

```
implyR('R=="v^2+w^2<=r^2->[{v'=w,w'=-v}]v^2+w^2<=r^2");
dIRule('R=="[{v'=w,w'=-v}]v^2+w^2<=r^2"); <(
  "dI Init": id,
  "dI Step":
     Dassignb('R=="[w':=-v;][v':=w;]2*v^(2-1)*v'+2*w^(2-1)*w'<=0");
     Dassignb('R=="[v':=w;]2*v^(2-1)*v'+2*w^(2-1)*(-v)<=0");
     QE
)
```

Because the usual steps after dI are almost always the same, there also is a shorter tactic achieving the same thing where `dIClose` already follows up `dI` with the usual assignments and real arithmetic:

```
implyR(1) ; dIClose(1)
```

Here is the same tactic listing the formulas explicitly rather than their positions:

```
implyR('R=="v^2+w^2<=r^2->[{v'=w,w'=-v}]v^2+w^2<=r^2");
dIClose('R=="[{v'=w,w'=-v}]v^2+w^2<=r^2")
```

Of course, an automatic proof by the tactic **Auto** would have worked as well.

Example 4.4 (Damped oscillator)**.** The differential equation $x' = y, y' = -\omega^2 x - 2d\omega y$ describes the damped oscillator with the undamped angular frequency ω and damping ratio d. It always stays in the region $\omega^2 x^2 + y^2 \le c^2$ as expressed by this dL formula:

$$\omega^2 x^2 + y^2 \le c^2 \to [x' = y, y' = -\omega^2 x - 2d\omega y \,\&\, (\omega \ge 0 \land d \ge 0)]\, \omega^2 x^2 + y^2 \le c^2$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
w^2*x^2+y^2<=c^2 -> [{x'=y,y'=-w^2*x-2*d*w*y & w>=0&d>=0}] w^2*x^2+y^2<=c^2
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential invariance rule dI. The initial case in the left premise proves by **Prop**. The differential case in the right premise proves after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with ℝ via **Tools->Real Arithmetic**. This gives you the following proof where the premise for the initial case is elided to save some space:

$$
\dfrac{\dfrac{\dfrac{\dfrac{*}{\omega \geq 0 \land d \geq 0 \vdash 2\omega^2 xy - 2\omega^2 xy - 4d\omega y^2 \leq 0}\ \mathbb{R}}{\omega \geq 0 \land d \geq 0 \vdash [x' := y][y' := -\omega^2 x - 2d\omega y] 2\omega^2 x x' + 2yy' \leq 0}\ [:=]}{\omega^2 x^2 + y^2 \leq c^2 \vdash [x' = y, y' = -\omega^2 x - 2d\omega y \,\&\, \omega \geq 0 \land d \geq 0]\, \omega^2 x^2 + y^2 \leq c^2}\ \mathrm{dI}}{\vdash \omega^2 x^2 + y^2 \leq c^2 \to [x' = y, y' = -\omega^2 x - 2d\omega y \,\&\, \omega \geq 0 \land d \geq 0]\, \omega^2 x^2 + y^2 \leq c^2}\ \to\mathrm{R}
$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; dIRule(1) ; <(
  propClose,
  Dassignb(1) ; Dassignb(1) ; QE
  )
```

The same proof can be conducted with explicitly listing the formulas that the tactics are applied to rather than their positions:

```
implyR('R=="w^2*x^2+y^2<=c^2->[{x'=y,y'=-w^2*x-2*d*w*y&w>=0&d>=0}]w^2*x^2+y^2<=c^2")
dIRule('R=="[{x'=y,y'=-w^2*x-2*d*w*y&w>=0&d>=0}]w^2*x^2+y^2<=c^2"); <(
  "dI Init": propClose,
  "dI Step":
    Dassignb('R=="[y':=-w()^2*x-2*d*w()*y;][x':=y;]w()^2*(2*x^(2-1)*x')+2*y^(2-1)*y'
    Dassignb('R=="[x':=y;]w()^2*(2*x^(2-1)*x')+2*y^(2-1)*(-w()^2*x-2*d*w()*y)<=0");
    QE
)
```

Of course, an automatic proof by **Auto** would have worked as well.

Example 4.5 (Lots of squares, attempted naively)**.** Consider the following simple dL formula:

$$x \geq 1 \land y \geq 0 \to [x' = x^2 + y, y' = y^2]\, x \geq 1$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>=1&y>=0->[{x'=x^2+y,y'=y^2}]x>=1
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential invariance rule dI. The initial case in the left premise proves by **Auto**. The canonical steps in the differential case in the right premise are to use the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with ℝ via **Tools->Real**

Arithmetic, where a problem comes up. This gives you the following proof attempt:

$$\dfrac{\dfrac{\dfrac{*}{x\geq0\land y\geq0 \vdash x\geq 0}\text{id} \qquad \dfrac{\dfrac{\dfrac{\lightning\ \mathit{false}}{\vdash x^2+y\geq 0}\mathbb{R}}{\vdash [y':=y^2]x^2+y\geq 0}[:=]}{\vdash [x':=x^2+y][y':=y^2]x'\geq 0}[:=]}{x\geq 1\land y\geq 0 \vdash [x'=x^2+y, y'=y^2]x\geq 1}\text{dI}}{\vdash x\geq 1\land y\geq 0 \rightarrow [x'=x^2+y, y'=y^2]x\geq 1}\rightarrow\text{R}$$

The remaining premise $\vdash \mathit{false}$ for the differential case is impossible to prove (although if you did the impossible and prove it, it would correctly imply its conclusion $\vdash x^2+y\geq 0$). The differential case is *not provable*, because $x^2+y\geq 0$ is not true in all states. Indeed, **Tools->Counterexample** reports a counterexample:

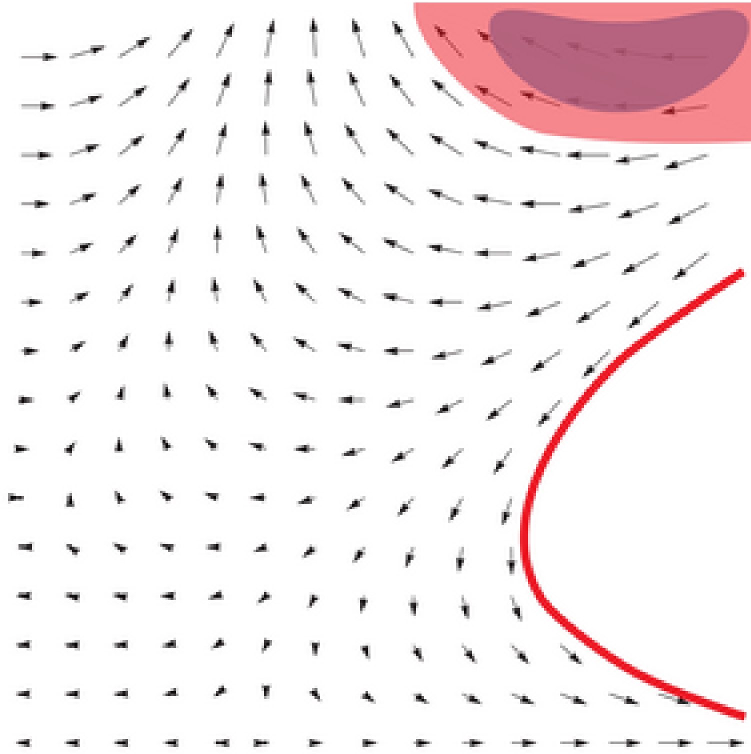
x	0
y	-1

Indeed, if y were negative, which is perfectly allowed in the affected sequent $\vdash x^2+y\geq 0$, the formula would evaluate to false. Now the initial state knew that $y\geq 0$ initially, but the question is whether that remains true when the ODE is evolving. That is why we first need to establish that $y\geq 0$ is an invariant, which is what the next section will investigate.

4.4 Differential Cuts

Differential cuts. Differential cuts accumulate knowledge about the evolution of an ODE from multiple proofs. The differential cut rule dC is for lemmas in the middle of the differential equation. It enables you to first prove a postcondition C of a differential equation on one branch and then assume C in the evolution domain constraint of the differential equation on the other branch. This establishes a property C as a lemma about an ODE that you then get to assume from now on in the middle of the differential equation:

$$\text{dC}\ \frac{\Gamma\vdash [x'=f(x)\,\&\,Q]C,\Delta \quad \Gamma\vdash [x'=f(x)\,\&\,(Q\land C)]P,\Delta}{\Gamma\vdash [x'=f(x)\,\&\,Q]P,\Delta}$$



Differential cuts are useful when successively identifying auxiliary invariants that are easier to prove than the original postcondition P but ultimately help establish P.

Example 4.6 (Lots of squares)**.** Consider the following simple dL formula:

$$x \geq 1 \land y \geq 0 \to [x' = x^2 + y, y' = y^2]\, x \geq 1$$

A direct proof with differential invariants dI does not succeed (see 4.5), because $y \geq 0$ first needs to be shown to be invariant. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>=1&y>=0->[{x'=x^2+y,y'=y^2}]x>=1
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential cut rule dC then insert `y>=0` for R and apply dC. On the branch that uses the differential cut $y \geq 0$ in the evolution domain constraint, select the augmented differential equation $[x' = x^2 + y, y' = y^2 \,\&\, y \geq 0]\, x \geq 1$ with a **right-click** and use the differential invariant rule dI. The initial case proves automatically or by **Prop**. The differential case proves automatically or after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with ℝ via **Tools->Real Arithmetic**. On the branch that shows the differential cut $y \geq 0$ proceed likewise: select the differential equation $[x' = x^2 + y, y' = y^2]\, y \geq 0$ with a **right-click** and use the differential invariant rule dI. The initial case proves automatically or by **Prop**. The differential case proves automatically or after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with ℝ via **Tools->Real Arithmetic**. This gives you the following

proof where the premises for the initial cases are elided to save some space:

$$
{\scriptstyle \rightarrow R}\dfrac{{\scriptstyle dC}\dfrac{{\scriptstyle dI}\dfrac{{\scriptstyle [:=]}\dfrac{{\scriptstyle [:=]}\dfrac{{\scriptstyle \mathbb{R}}\dfrac{*}{\vdash y^2 \geq 0}}{\vdash [y':=y^2]y' \geq 0}}{\vdash [x':=x^2+y][y':=y^2]y' \geq 0}}{x\geq 1 \land y \geq 0 \vdash [x'=x^2+y, y'=y^2]y \geq 0} \quad {\scriptstyle dI}\dfrac{{\scriptstyle [:=]}\dfrac{{\scriptstyle [:=]}\dfrac{{\scriptstyle \mathbb{R}}\dfrac{*}{y \geq 0 \vdash x^2+y \geq 0}}{y \geq 0 \vdash [y':=y^2]x^2+y \geq 0}}{y \geq 0 \vdash [x':=x^2+y][y':=y^2]x' \geq 0}}{x\geq 1 \land y \geq 0 \vdash [x'=x^2+y, y'=y^2 \,\&\, y \geq 0]x \geq 1}}{x \geq 1 \land y \geq 0 \vdash [x'=x^2+y, y'=y^2]x \geq 1}}{\vdash x \geq 1 \land y \geq 0 \rightarrow [x'=x^2+y, y'=y^2]x \geq 1}
$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; dC("y>=0", 1) ; <(
  dIRule(1) ; <(
    propClose,
    Dassignb(1) ; Dassignb(1) ; QE
    ),
  dIRule(1) ; <(
    propClose,
    Dassignb(1) ; Dassignb(1) ; QE
    )
  )
```

Here is the same tactic that explicitly lists the affected formulas:

```
implyR('R=="x>=1&y>=0->[{x'=x^2+y,y'=y^2}]x>=1");
dC("y>=0", 'R=="[{x'=x^2+y,y'=y^2}]x>=1"); <(
  "Use": dIRule('R=="[{x'=x^2+y,y'=y^2&true&y>=0}]x>=1"); <(
      "dI Init": propClose,
      "dI Step":
        Dassignb('R=="[y':=y^2;][x':=x^2+y;]x'>=0");
        Dassignb('R=="[x':=x^2+y;]x'>=0");
        QE
    ),
  "Show": dIRule('R=="[{x'=x^2+y,y'=y^2}]y>=0"); <(
      "dI Init": propClose,
      "dI Step":
        Dassignb('R=="[y':=y^2;][x':=x^2+y;]y'>=0");
        Dassignb('R=="[x':=x^2+y;]y^2>=0");
        QE
    )
)
```

Using the tactic `dIClose` for the usual follow-up after dI using the slower real arithmetic instead of propositional reasoning, this tactic simplifies as follows:

```
implyR(1) ; dC("y>=0", 1) ; <(
  dIClose(1),
  dIClose(1)
  )
```

Of course, an automatic proof by **Auto** would also have worked, too, but you would not have learned much about how it works and what you can do to help in complicated cases beyond the reach of automatic verification.

4.5 @Invariant Annotation

Important (Differential @invariant annotation). Differential invariants are often more complicated than their postconditions, because other crucial information on the historical system behavior needs to be transported through the proof, as you have already seen in Chapter 11: Differential Equations & Proofs. Thus, once you have found the sequence of differential invariants to use for differential cuts (see 4.4), it is good practice to write them directly into the hybrid program. Not only will this make sure KeYmaera X does not need to ask you for it again and avoids expensive differential invariant search procedures, but it will also help you understand your system better in the future. That understanding of the differential invariants is particularly helpful when you change your system design with additional control cases. If you have written down the differential invariants, then you know what needs to be preserved when you change the controller. To record the differential invariants for your differential equation, annotate it after the differential equation system and its evolution domain constraint with an `@invariant(C)` annotation that remembers the loop invariant C to use. For example, write something like:

```
{x'=f(x) & Q}@invariant(magicFormula)
```

If you have a sequence of differential cuts, you can annotate the list of differential invariants in the order that they are cut in:

```
{x'=f(x) & Q}@invariant(magicFormula1,magicFormula2,magicFormula3)
```

This means the first differential cut uses `magicFormula1` after which the next differential cut uses `magicFormula1` and the last differential cut is `magicFormula3`.

Example 4.7 (Differential invariant annotations for lots of squares)**.** The differential invariants that are used for differential cuts in 4.6 are the most important ingredient in their proof. You should make it a habit to write down the differential invariants that you differential cut in as part of the hybrid program. That speeds up proof search and makes your job easier, too, when you change the system design later.

```
x>=1&y>=0->[{x'=x^2+y,y'=y^2}@invariant(y>=0)]x>=1
```

Load this example in KeYmaera X and use **Auto** to prove it. Admittedly, this worked just as well before the differential invariant annotation, but then it was using differential invariant search, which can take more time or time out.

4.6 Differential Weakening

Differential weakening. The easiest proof rule for differential equations $x' = f(x) \,\&\, Q$ proves properties directly just from the evolution domain constraint Q. After all, the system can only follow $x' = f(x) \,\&\, Q$ while the evolution domain constraint Q is true.

$$\text{dW}\ \frac{\Gamma_{\text{const}}, Q \vdash P, \Delta_{\text{const}}}{\Gamma \vdash [x' = f(x) \,\&\, Q]P, \Delta}$$

where Γ_{const} and Δ_{const} is the constant part of Γ, Δ, respectively, which are only formulas that have no free variables that are bound in the differential equation $x' = f(x) \& Q$. Hence, formulas of Γ, Δ that only mention constants or variables other than x, x' will remain in $\Gamma_{\text{const}}, \Delta_{\text{const}}$. Formulas in Γ that mention x such as $x > 0$, obviously, cannot be kept in Γ_{const}, soundly, because, unlike Q, they are not known to still be true after the ODE. The situation for Δ_{const} is accordingly.

Example 4.8 (Bouncing balls stay above ground)**.** Consider the differential equation $x' = v, v' = -g \& x \geq 0$ for the bouncing ball at height x falling with velocity v in gravity $g > 0$ above ground, so within the evolution domain constraint $x \geq 0$. It is, indeed, true (although not a revolutionary insight) that this differential equation that is restricted to remain above ground indeed always is above ground. That is expressed by the following dL formula for a bouncing ball starting at initial height 5:

$$x = 5 \land g > 0 \rightarrow [x' = v, v' = -g \& x \geq 0]\, 0 \leq x$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x=5 & g>0 -> [{x'=v,v'=-g&x>=0}] 0<=x
```

Load this example in KeYmaera X and **left-click** to use $\rightarrow$R to move the assumption into the antecedent, then split the conjunction in the antecedent by a **left-click** to use $\land$L, then **right-click** on the differential equation to select the differential weakening rule dW. The resulting arithmetic proves by real arithmetic arithmetic with $\mathbb{R}$ via **Tools->Real Arithmetic**. This gives you the following proof:

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{g > 0, x \geq 0 \vdash 0 \leq x}\;\mathbb{R}}{x = 5, g > 0 \vdash [x' = v, v' = -g \& x \geq 0]\, 0 \leq x}\;\text{dW}}{x = 5 \land g > 0 \vdash [x' = v, v' = -g \& x \geq 0]\, 0 \leq x}\;\land\text{L}}{\vdash x = 5 \land g > 0 \rightarrow [x' = v, v' = -g \& x \geq 0]\, 0 \leq x}\;\rightarrow\text{R}$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; andL(-1) ; dW(1) ; QE
```

The same proof with explicit mention of the formulas that the tactics are applied to is as follows:

```
implyR('R=="x=5&g>0->[{x'=v,v'=-g&x>=0}]0<=x");
andL('L=="x=5&g>0");
dW('R=="[{x'=v,v'=-g&x>=0}]0<=x");
QE
```

Of course, an automatic proof by **Auto** would have worked as well. KeYmaera X is usually clever enough to decompose the context Γ, Δ to read off the most informative constant part $\Gamma_{\text{const}}, \Delta_{\text{const}}$ even if formulas have not been split perfectly yet. But using $\land$L suitably to split it in a controlled way is a good idea for clarity and efficiency.

Differential weakening after cut. In and of itself, differential weakening rule dW is quite impoverished, because it completely discards the differential equation and only proves

properties straight from the evolution domain constraint. If that is possible for your original model, you should take note that you are assuming incredibly strong evolution domain constraints in your model, and scrutinize those assumptions on the behavior of physics. But differential weakening rule dW can also become perfectly useful after differential cuts by rule dC have augmented the evolution domain constraint to become informative.

4.7 Conserved Quantities

Conserved quantities. Even if you are really interested in proving something else, it can often be magnificently useful to first prove something entirely different and then use it as a helpful assumption for the original question. By far the most helpful and fundamental invariants for that are conserved quantities. These are terms e that never change their value when following the dynamics. That is, whatever real value the term e has before the differential equation, term e still has exactly the same real value after following the differential equation for any amount of time. That is, the following formula is true, where e_0 is an extra variable remembering the initial value of e:

$$e = e_0 \rightarrow [x' = f(x)]\, e = e_0$$

If the value of the particular term e has the real value of e_0 initially, then it always will have that value, no matter how long we follow the ODE $x' = f(x)$.

This is a conceptually straightforward yet exceedingly powerful concept. The need to dream up a variable or constant function symbols e_0 to remember the initial value is somewhat annoying. At the same time, the knowledge that term e remains constant is used so frequently in establishing other properties, that it is best to use the @invariant annotation from 4.5 in the following form:

```
{x'=f(x) & Q}@invariant(e=old(e))
```

The special symbol `old()` is understood by KeYmaera X to indicate the value that its argument term had before the ODE. Hence, the `@invariant` annotation `e=old(e)` precisely indicates that the term `e` will always equal its old, initial value. That is, term `e` is a conserved quantity of the dynamics. As usual with `@invariant` annotations, KeYmaera X will go right ahead and prove, by a differential cut, that `e` indeed never changes its value along the ODE and will then give you that knowledge for the remaining proof. Because conserved quantities `e` fundamentally link the values of the respective variables of the differential equation, they give you exceedingly strong information that you can rely on when conducting your proof or designing your system.

Example 4.9 (Conserved quantities for rotational dynamics)**.** Consider a minor variation of the dynamics from 4.2 for the rotation of the point v, w on a circle of some unspecified radius around the origin with conjecture that w is always nonzero whenever v is zero:

$$v \neq 0 \rightarrow [v' = w, w' = -v](v = 0 \rightarrow w \neq 0)$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
v!=0 -> [{v'=w,w'=-v}](v=0 -> w!=0)
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent. Resist the temptation to **left-click** and select the automatic `ODE` tactic, because that would just prove the question automatically. Instead, think about what invariant would help you prove this property.

Directly trying to prove it as a differential invariant is unpromising, because the rate of change of w is $-v$. But this ODE has a conserved quantity: even if v and w change, $v^2 + w^2$ will always have the same value when following the ODE. By assumption the initial v is nonzero, so the initial $v^2 + w^2$, which, as a conserved quantity, equals the value of $v^2 + w^2$ after the ODE, is also nonzero. Hence, if v is zero, w cannot possibly be zero. Following this proof certainly needs such an indirect argument, but conserved quantities make it conceptually fairly straightforward.

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR('R=="v!=0->[{v'=w,w'=-v}](v=0->w!=0)");
dC("v^2+w^2=old(v^2+w^2)", 'R=="[{v'=w,w'=-v}](v=0->w!=0)"); <(
  "Use":
    dW('R=="[{v'=w,w'=-v&true&v^2+w^2=old}](v=0->w!=0)");
    QE,
  "Show":
    dIRule('R=="[{v'=w,w'=-v}]v^2+w^2=old"); <(
      "dI Init":
        QE,
      "dI Step":
        Dassignb('R=="[w':=-v;][v':=w;]2*v^(2-1)*v'+2*w^(2-1)*w'=0");
        Dassignb('R=="[v':=w;]2*v^(2-1)*v'+2*w^(2-1)*(-v)=0");
        QE
    )
)
```

Of course, a more succinct proof relying mostly on proof automation has the same effect:

```
implyR(1) ; dC("v^2+w^2=old(v^2+w^2)", 1) ; <(
  dIClose(1),
  dIClose(1)
  )
```

The key insight behind this proof is the identification of the conserved quantity, which is best attached with an `@invariant(...)` annotation right into the model for the purposes of documentation and speeding up proof automation:

```
v!=0 -> [{v'=w,w'=-v}@invariant(v^2+w^2=old(v^2+w^2))](v=0 -> w!=0)
```

Important (Conserved quantities with const()). Since conserved quantities are fundamental for differential equations and hybrid systems, KeYmaera X provides special support to indicate that a term `e` is a conserved quantity:

```
{x'=f(x)}@invariant(e=const())
```

This instructs KeYmaera X to prove that the term `e` is a conserved quantity, so never changes its value while following the ODE `x'=f(x)` for any duration. The above annotation is short for

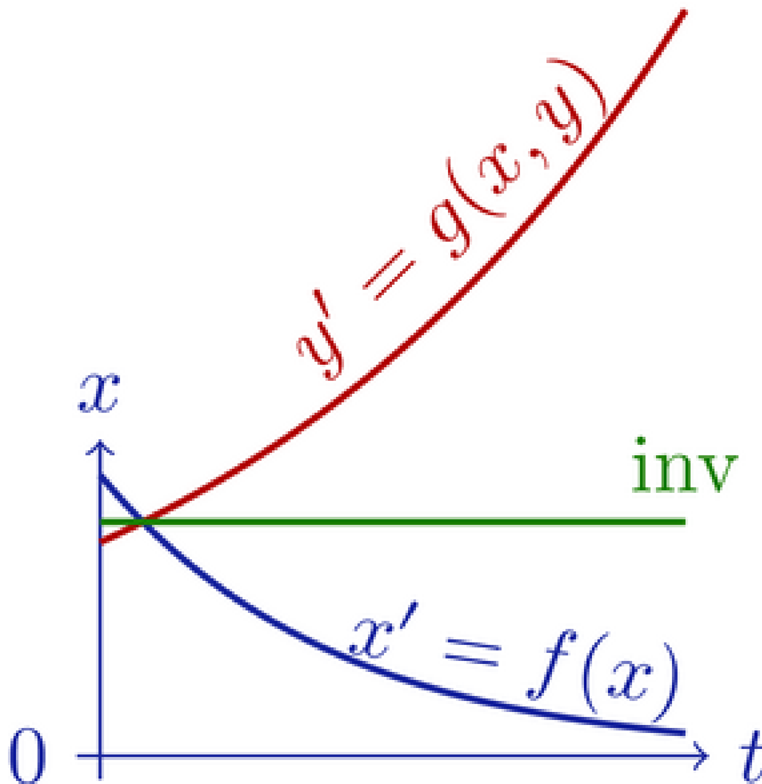
```
{x'=f(x)}@invariant(e=const(e))
```

The use of `const()` is not much of a saving, except for long and complicated terms `e`, but is still preferred for clear communication. It also fits to common conventions in physics to indicate that a quantity is constant.

4.8 Differential Ghosts

Differential ghosts. Differential cuts make it possible to successively accumulate provable information about differential equations until the original conjecture can be proved from accumulated knowledge. But the individual invariants that are differentially cut-in still need to be proved. This is especially problematic when the "trend of truth" is towards false, i.e., the property is getting less true over time. *Differential ghosts* add differential equations for new ghost variables to the existing system of differential equations enabling reasoning about the historical evolution of ODE systems in integral form. The differential ghosts proof rule can add a new differential equation $y' = a(x) \cdot y + b(x)$ into the differential equation system:

$$\text{dG}\ \frac{\Gamma \vdash \exists y\, [x' = f(x), y' = a(x) \cdot y + b(x)\,\&\, Q]P, \Delta}{\Gamma \vdash [x' = f(x)\,\&\, Q]P, \Delta}$$



In and of itself, this differential ghost rule only increases the dimension of the differential equation. To benefit from that additional dimension, rule dG is typically used in

combination with rephrasing the postcondition P to a new postcondition G in a way that relates the new differential ghost variable y with the system state variables x:

$$\text{dG}'\ \frac{\Gamma \vdash \exists y\,[x' = f(x), y' = a(x) \cdot y + b(x)\,\&\,Q]G, \Delta \quad G \vdash P}{\Gamma \vdash [x' = f(x)\,\&\,Q]P, \Delta}$$

This rule is used when you apply dG and click on the postcondition P to edit it.

Example 4.10 (Trying exponential decay, naively, with dI)**.** Consider the exponential decay example, which this example will first try to prove naively by differential invariants dI:

$$x > 0 \rightarrow [x' = -x]\,x > 0$$

Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>0->[{x'=-x}]x>0
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential invariance rule dI. The initial case in the left premise proves by id via **Propositional->Identity**. The differential case in the right premise proves after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with $\mathbb{R}$ via **Tools->Real Arithmetic**, where a problem comes up. This gives you the following proof attempt:

$$\dfrac{\dfrac{\dfrac{*}{x > 0 \vdash x > 0}\mathbb{R} \qquad \dfrac{\dfrac{\dfrac{\lightning\ \mathit{false}}{\vdash -x \geq 0}\mathbb{R}}{\vdash [x' {:=} {-x}]\,x' \geq 0}[:=]}{}}{x > 0 \vdash [x' = -x]\,x > 0}\text{dI}}{\vdash x > 0 \rightarrow [x' = -x]\,x > 0}\rightarrow\text{R}$$

Here is the corresponding Bellerophon tactic performing the same proof attempt:

```
implyR(1) ; dIRule(1) ; <(
 id,
 Dassignb(1) ; QE
 )
```

The remaining premise $\vdash \mathit{false}$ for the differential case is impossible to prove (although if you did the impossible and prove it, it would correctly imply its conclusion $\vdash -x \geq 0$). The differential case is *not provable*, because $-x \geq 0$ is not true in all states, in fact, it is not even true in the initial state where $x > 0$. Indeed, **Tools->Counterexample** reports a counterexample:

x	1

Intuitively, this shows that the trend of the truth value of the postcondition $x > 0$ is towards getting false over time. And, indeed, the dynamics is getting closer and closer to violating $x > 0$, which makes it a bit of a miracle that it still stays above 0. The fact that the trend $-x$ is negative even in the initial state is an intuitive indication that there also is no differential cut that would help prove it. Indeed, it is provable that no combination of differential invariants and differential cuts prove this valid formula. Fortunately, differential ghosts dG are perfectly capable of proving the dL formula regardless. Of course, the proof technique of differential ghosts does not only work for exponential decay, but generalizes significantly (See Chapter 12 and JACM'20).

Example 4.11 (Exponential decay, explicit witness dG)**.** Consider the exponential decay example in which the dynamics is getting closer and closer to violating $x > 0$:

$$x > 0 \to [x' = -x]\, x > 0$$

Intuitively because of this trend towards violating the postcondition is a direct differential invariants proof impossible, also when using differential cuts. Instead, a differential ghost $y' = \frac{1}{2}y$ makes it possible to rephrase postcondition $x > 0$ to the equivalent $xy^2 = 1$ that is a differential invariant of the augmented dynamics. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>0->[{x'=-x}]x>0
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential ghost rule dG then insert the differential ghost `y'=(1/2)*y` for `E` in the differential equation and insert `x*y^2=1` for the new postcondition G and apply dG. Then **right-click** the resulting existential quantifier $\exists y$ and insert the witness `1/x^0.5` for θ in rule ∃R, which you are allowed to do as $x > 0$ by assumption. In the remaining differential equation **right-click** and use differential invariants dI. Then do the usual follow-up either automatically or manually. Prove the initial case with $\mathbb{R}$ via **Tools->Real Arithmetic**. Prove the differential case after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with $\mathbb{R}$ via **Tools->Real Arithmetic**. This gives you the following proof:

$$\dfrac{\dfrac{\dfrac{*}{x > 0, y = 1/x^{0.5} \vdash xy^2 = 1}\;\mathbb{R} \qquad \dfrac{\dfrac{*}{\vdash -xy^2 + 2xy\frac{y}{2} = 0}\;\mathbb{R}}{\vdash [x':=-x][y':=\frac{y}{2}]x'y^2 + x2yy' = 0}\;[:=]}{\dfrac{x > 0, y = 1/x^{0.5} \vdash [x' = -x, y' = \frac{y}{2}]\, xy^2 = 1}{\dfrac{x > 0 \vdash \exists y\, [x' = -x, y' = \frac{y}{2}]\, xy^2 = 1}{\dfrac{x > 0 \vdash [x' = -x]\, x > 0}{\vdash x > 0 \to [x' = -x]\, x > 0}\;\to\text{R}}\;\text{dG}}\;\exists\text{R}}\;\text{dI}}{}$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; dG("y'=1/2*y", "x*y^2=1", 1) ; existsR("1/x^0.5", 1) ; dIRule(1) ; <(
 QE,
 Dassignb(1) ; Dassignb(1) ; QE
 )
```

Here is the same proof when explicitly listing the formulas that the tactics are being applied to:

```
implyR('R=="x>0->[{x'=-x}]x>0");
dG("y'=1/2*y", "x*y^2=1", 'R=="[{x'=-x}]x>0");
existsR("1/x^0.5", 'R=="\exists y [{x'=-x,y'=1/2*y+0}]x*y^2=1");
dIRule('R=="[{x'=-x,y'=1/2*y+0}]x*y^2=1"); <(
  "dI Init": QE,
  "dI Step":
    Dassignb('R=="[y':=1/2*y+0;][x':=-x;]x'*y^2+x*(2*y^(2-1)*y')=0");
    Dassignb('R=="[x':=-x;]x'*y^2+x*(2*y^(2-1)*(1/2*y+0))=0");
```

```
    QE
)
```

An equivalent tactic uses the version of dI that directly closes the proof:

```
implyR(1) ; dG("y'=1/2*y", "x*y^2=1", 1) ; existsR("1/x^0.5", 1) ; dclose(1)
```

While this very explicit proof works, it has the downside of requiring the construction of an explicit witness for the initial value of y, which, here, was the unwieldy division by a square root $1/x^{0.5}$ that needs a fair amount of your attention to be well-defined.

Example 4.12 (Exponential decay, contextual dG)**.** Consider the exponential decay example in which the dynamics is getting closer and closer to violating $x > 0$:

$$x > 0 \rightarrow [x' = -x]\, x > 0$$

This proof will use the same differential ghost and rephrased postcondition as in 4.11 but proves its proof steps in context to avoid the need to name an explicit witness for the resulting existential quantifier. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>0->[{x'=-x}]x>0
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential ghost rule dG then insert the differential ghost `y'=(1/2)*y` for `E` in the differential equation and insert `x*y^2=1` for the new postcondition G and apply dG. Then **right-click** on the differential equation within the resulting existential quantifier $\exists y$ and use differential invariants dI. Then use the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with $\mathbb{R}$ via **Tools->Real Arithmetic**. This gives you the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x > 0 \vdash \exists y\, \big(xy^2 = 1 \land \forall x\, \forall y\, (-xy^2 + 2xy\frac{y}{2} = 0)\big)}\ \mathbb{R}}{x > 0 \vdash \exists y\, \big(xy^2 = 1 \land \forall x\, \forall y\, [x' {:=} -x][y' {:=} \frac{y}{2}]x'y^2 + x2yy' = 0\big)}\ [:=]}{x > 0 \vdash \exists y\, [x' = -x, y' = \frac{y}{2}]\, xy^2 = 1}\ \mathrm{dI}}{x > 0 \vdash [x' = -x]\, x > 0}\ \mathrm{dG}}{\vdash x > 0 \rightarrow [x' = -x]\, x > 0}\ \rightarrow\mathrm{R}$$

Here is the corresponding Bellerophon tactic performing the same proof:

```
implyR(1) ; dG("y'=1/2*y", "x*y^2=1", 1) ; dIRule(1.0) ; Dassignb(1.0.1.1.0.0) ; Das
```

The advantage of this proof style is that it retains full local information. It is easier to understand by working back directly from the axioms that KeYmaera X uses in context.

Example 4.13 (Exponential decay, implicit witness dG)**.** Consider the exponential decay example in which the dynamics is getting closer and closer to violating $x > 0$:

$$x > 0 \rightarrow [x' = -x]\, x > 0$$

This proof will use the same differential ghost and rephrased postcondition as in 4.11 but with an implicit technique to avoid the need to specify an explicit witness for its resulting existential quantifier. In the spirit of Bertrand Russell: The advantages of an implicit characterization over an explicit witness construction are roughly those of theft over honest toil. Here is the same differential dynamic logic formula transliterated to KeYmaera X:

```
x>0->[{x'=-x}]x>0
```

Load this example in KeYmaera X and **left-click** to use →R to move the assumption into the antecedent and **right-click** on the differential equation to select the differential ghost rule dG then insert the differential ghost `y'=(1/2)*y` for `E` in the differential equation and insert `x*y^2=1` for the new postcondition G and apply dG. Then **right-click** the resulting existential quantifier $\exists y$ and eliminate it by `existsRmon` with an implicit characterization inserting `x*y^2=1` for `g(x)`. In the remaining differential equation **right-click** and use differential invariants dI. Then do the usual follow-up either automatically or manually. Prove the initial case by id via **Propositional->Identity**. Prove the differential case after using the assignment axiom [:=] twice by a **left-click** or **right-click** followed by real arithmetic arithmetic with $\mathbb{R}$ via **Tools->Real Arithmetic**. This gives you the following proof:

$$
\dfrac{\dfrac{\dfrac{*}{x>0 \vdash \exists y\, xy^2=1}\ \mathbb{R} \qquad \dfrac{\dfrac{\dfrac{*}{\vdash -xy^2+2xy\frac{y}{2}=0}\ \mathbb{R}}{\vdash [x':=-x][y':=\frac{y}{2}]x'y^2+x2yy'=0}\ [:=]}{x>0, xy^2=1 \vdash [x'=-x, y'=\frac{y}{2}]\, xy^2=1}\ \text{dI}}{\dfrac{x>0 \vdash \exists y\, [x'=-x, y'=\frac{y}{2}]\, xy^2=1}{x>0 \vdash [x'=-x]\, x>0}\ \text{dG}}\ ??}{\vdash x>0 \to [x'=-x]\, x>0}\ \to\text{R}
$$

Here is the corresponding Bellerophon tactic performing the same proof, while eliding the premise for the initial case of dI for brevity:

```
implyR(1) ; dG("y'=1/2*y", "x*y^2=1", 1) ; existsRmon("x*y^2=1", 1) ; <(
  QE,
  dIRule(1) ; <(
    id,
    Dassignb(1) ; Dassignb(1) ; QE
    )
  )
```

Here is the same proof with explicit formulas that tactics are being applied to:

```
implyR('R=="x>0->[{x'=-x}]x>0");
dG("y'=1/2*y", "x*y^2=1", 'R=="[{x'=-x}]x>0");
existsRmon("x*y^2=1", 'R=="\exists y [{x'=-x,y'=1/2*y+0}]x*y^2=1"); <(
  "Use": QE,
  "Show": dIRule('R=="[{x'=-x,y'=1/2*y+0}]x*y^2=1"); <(
      "dI Init": id,
      "dI Step":
        Dassignb('R=="[y':=1/2*y+0;][x':=-x;]x'*y^2+x*(2*y^(2-1)*y')=0");
        Dassignb('R=="[x':=-x;]x'*y^2+x*(2*y^(2-1)*(1/2*y+0))=0");
        QE
    )
)
```

An equivalent tactic uses the version of dI that directly closes the proof:

```
implyR(1) ; dG("y'=1/2*y", "x*y^2=1", 1) ; existsRmon("x*y^2=1", 1) ; <(
  QE,
```

```
dIClose(1)
)
```

Bibliography

[1] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham, 2018. URL: http://www.springer.com/978-3-319-63587-3, doi:10.1007/978-3-319-63588-0.

[2] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 59(2):219–265, 2017. doi:10.1007/s10817-016-9385-1.

[3] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In Amy Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538, Berlin, 2015. Springer. doi:10.1007/978-3-319-21401-6_36.

[4] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010. URL: http://www.springer.com/978-3-642-14508-7, doi:10.1007/978-3-642-14509-4.

[5] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008. doi:10.1007/s10817-008-9103-8.

[6] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. Bellerophon: Tactical theorem proving for hybrid systems. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP*, volume 10499 of *LNCS*, pages 207–224. Springer, 2017. doi:10.1007/978-3-319-66107-0_14.

[7] André Platzer. Logics of dynamical systems. In LICS [15], pages 13–24. doi:10.1109/LICS.2012.13.

[8] André Platzer. The complete proof theory of hybrid systems. In LICS [15], pages 541–550. doi:10.1109/LICS.2012.64.

[9] André Platzer and Yong Kiam Tan. Differential equation invariance axiomatization. *J. ACM*, 67(1):6:1–6:66, 2020. doi:10.1145/3380825.

[10] André Platzer. Differential game logic. *ACM Trans. Comput. Log.*, 17(1):1:1–1:51, 2015. doi:10.1145/2817824.

[11] Stefan Mitsch and André Platzer. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1-2):33–74, 2016. Special issue of selected papers from RV'14. doi:10.1007/s10703-016-0241-z.

[12] Andrew Sogokon, Stefan Mitsch, Yong Kiam Tan, Katherine Cordwell, and André Platzer. Pegasus: Sound continuous invariant generation. *Form. Methods Syst. Des.* Special issue for selected papers from FM'19. doi:10.1007/s10703-020-00355-z.

[13] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. Formally verified differential dynamic logic. In Yves Bertot and Viktor Vafeiadis, editors, *Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017*, pages 208–221, New York, 2017. ACM. doi:10.1145/3018610.3018616.

[14] André Platzer. *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2008.

[15] *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, Los Alamitos, 2012. IEEE.