KeYmaera X^{++}
Improving the Proof Experience

Corwin de Boor
cdeboor@andrew.cmu.edu

December 7, 2018

Abstract

Because hybrid systems describe both discrete control and continuous dynamics, they are attractive for modeling computer systems that interact with the real world. To ensure that these dangerous systems do not injure anyone, we would like to verify their controllers formally. We present improvements to the hybrid system verification tool KeYmaera X [4] to facilitate better modes of user interaction. By saving proof states in the middle of execution and linking the tactic language visually with the current goals, we enable more effective proof construction and better proof introspection.

1 Introduction

Self-driving cars, airplanes, and even surgery assistants are all cyber-physical systems: systems that mix discrete logic control with the continuous dynamics of the real world. Many of the systems that fall under this umbrella require careful design because any control errors can result in death. Humans make errors, so trusting that the system designers and engineers managed to think of all corner cases is too much of a risk.

Instead of needing to trust any particular person, we would rather trust a formal proof of correctness of a hybrid system. To this end, $\mathsf{d}\mathcal{L}$ provides a language for modeling these systems in a way that their safety can be verified. KeYmaera X is a proof assistant for showing properties of hybrid systems modeled in $\mathsf{d}\mathcal{L}$ Mitsch and Platzer [4]. Because KeYmaera X only allows you to construct true statements, the only component you need to trust is the core.

Unfortunately, the proof construction interface is is sequential. To construct a proof, you apply sound proof rules and axioms one after the other. Each step modifies your goals, giving you new tasks to work on until the proof is complete.

This style of interaction works well when you know for a fact that your statement is valid and you know exactly how the proof should be structured. For the same reason we need proof verification help, we also need proof construction help: people make mistakes. Often times, you makes a mistake in your model and you need to start the proof over from the beginning. Other times, your model is correct but you make a crucial error early on in your proof. To rectify that error, you need to either start over from the beginning or step backwards, undoing each of your applied rules one at a time, until you reach the point of error.

This situation is partially helped Bellerophon, a language of proof tactics for KeYmaera X [3]. Using this tactic language, you can copy your current tactic, rewind to the point of error, fix the error, and replay the tactic to get you back to where you were.

One major issue with this method of fixing mistakes is inherent to the types of programs expressible in $\mathsf{d}\mathcal{L}$. The programs are fundamentally non-discrete which makes reasoning about their behavior more difficult. Real arithmetic is decidable, but it is not decidable quickly. KeYmaera X takes a very long time to verify many real arithmetic facts and therefore replaying tactics is often not a desirable strategy.

Another major issue with this approach results from the brittleness of Bellerophon. Most tactics involve applying a particular axiom to a particular formula in the sequent. Bellerophon tactics specify their target formulas with integer arguments that represent indices within the sequent. That means if a sequent is modified by changing an earlier part of the proof, the later tactics will operate on the wrong formula.

The brittleness of Bellerophon combines with another major issue of the KeYmaera X: all-or-nothing tactic execution. If you run a big tactic in the editor, you risk having your tactic indices slightly off and breaking the entire execution. The irony of this situation is later parts of the proof are much more likely to have tactic index bugs, but are also much more painful to fix. This rigidity provokes system verifiers to run their tactics one by one, to avoid repeated long wait times of having to fix tactics near the end of the proof. Now that we are running tactics one by one or a few at a time, the benefit of the tactic language for fixing mistakes is essentially nullified.

Because the sequents for hybrid systems are very large, one can only really view a couple sequents at a time. Running multi-step tactics provides additional confusion because we only are able to see the remaining open goals. Especially when using non-deterministic choice, many of the sequents look the same and figuring out which one corresponds to which branch of the proof is tricky. The tactic records what exactly happened in the proof, but its different format is difficult to connect to the derivation.

Contributions This work attempts to address or alleviate the above issues by improving the KeYmaera X proof experience. The first helping tool is a proof listener that records each step of the proof as it is executed. This listener also enables interactive execution where the interpreter effectively waits for the user to fix their bugs by storing pending tactics containing the tactic parts that failed. Then, additional tactic metadata is sent to the client enabling a closer relationship between the proof and the tactics. Now, the tactic editor highlights the sequence of tactics leading to the active goal in the proof, linking from goals to tactics; also, selecting tactics will show the sequent at their time of execution, linking from tactics to goals. Finally, a revamped tactic patch executor smooths the whole package by minimally pruning the tactic tree to minimize unnecessary re-computation.

2 Approach

To alleviate the combined pain of the brittleness of Bellerophon and the complexity of deciding real arithmetic, we created a the step-by-step proof listener. This listener saves the state of the proof into the database after the execution of each subtactic. Then, any errors stop after progress has been made, rather than requiring a complete re-execution. Because users are punished less by writing incorrect tactics, they are more free to perform nuanced interactions with the prover. This listener is described in detail in Section 3.

Writing a listener to save incremental progress into the database is made difficult by the need to separate the soundness-critical core and the tactic interpreter from the interactions with the database. To enable this separation of concerns, the Bellerophon interpreter invokes listener callbacks recursively upon starting and finishing the execution of a tactic. As the listener is fed a stream of tactic events, it needs to construct the corresponding derivation tree.

At a first glance, it seems fine to store the tactic start events into the database and update them into finish events after completion. However, what we want to show the user is actually the derivation, not the tactic that created it. The events that are fed to the listener actually correspond to an in-order traversal of the *tactic syntax tree*, not the the derivation tree that we want. As such, the listener needs to intelligently do an online tree-transformation, saving a derivation into the database from the tactic event stream.

To augment the data stored into the database, we created a new kind of tactic: the pending tactic. The pending tactic stores as its argument a tactic that has yet to be executed. When the step-by-step listener finds a tactic that encountered an error, it continues forward and saves the broken tactic as a pending tactic. Now, the user knows what they need to fix from the error message and knows where to fix it from the pending tactic.

To resolve the disconnect between the open goals and the tactics, we now generate additional tactic metadata. For each executed tactic, we store where in the tactic string it occurs. This enables us to look up a tactic within the string to highlight it or to find the tactic under the cursor to provide information about it. This tactic metadata and how it helps are described in greater detail in Section 4.

After running a large tactic, the user is presented a completely refreshed interface with new goals available; there was no link between the goals and how they appeared. To resolve this confusion, we now highlight all the tactics on the path through the derivation tree to the goal. This connects the goal back to how we got there so that there is less confusion about which goal corresponds to which part of the proof.

We also use the metadata to connect in the reverse direction. Given a tactic within the tactic trace, we can show the sequent that tactic was applied to. This enables us to look backwards in the history of the tactic execution to gain context about the proof. Rather than having to un-apply the tactics in your mind, you can just peek at what happened before to help figure out what to do next.

Now that we can figure out where we are and what we need to fix, we need to actually be able to edit the tactics. We discuss the tactic editing process in Section 5. When editing tactics, we would like to minimize the number of tactics that need to be run again. To do this, we compare the original tactic with the desired goal tactic. We find the least common ancestor of all of the changes, prune the tree at that node, and apply the updates below it. Choosing the least common ancestor ensures that we replay as few nodes as possible.

3 Tactic Tree Transformation

3.1 The Derivation Tree

The purpose of the tactic tree transformation is to convert the syntax tree of the tactics into the derivation tree stored in the database. We need to perform this transformation so that the database representation matches the kind of data that we would like to serve to users. As a given tactic is viewed much more than it is run (it runs only once), optimizing for the read operation is desirable.

Proofs are represented in the database as a tree of execution steps. An execution step stores a single rule application of the derivation tree. A single rule application corresponds to the combination of many tactics that, given a proof for several sequent subgoals, construct a proof for a single sequent conclusion. Thus our tree is an n-ary tree of rules.

To store the structure of the tree, each execution step maintains a parent pointer to the rule before it. This parent pointer is augmented with an index corresponding to which goal is being targeted. The goal index enables us to differentiate between the children of node: each child has a different goal index but the same parent.

At the end of the proof, every execution step either proves from no subgoals or has one child per subgoal. Our invariant about the construction of the tree then implies that we have a proof for the original desired formula. We can construct this whole proof by stitching together the individual deductions from each execution step.

To confuse matters slightly, the in-memory representation of the proof tree is slightly different. Rather than having nodes of the tree be the applied rules, nodes of the proof tree are edges between execution steps. The rationale behind the change in abstraction is we can apply tactics to a (not-yet created) edge to continue building the proof tree downwards. Alternatively, you can think of the proof tree nodes as sequents within the proof tree rather than the rules. Figure 1 shows an example of the relationship between the derivation, the in- memory proof tree, and the execution steps.

Each node stores its `maker`, the tactic that was run to build the node's corresponding sequent. Edges out of a node correspond essentially to the horizontal bars in a derivation that consume a bunch of subgoals to form a conclusion. This representation is a little bit confusing because all children of a single node have the same `maker`: the all came out of the same rule application. To figure out what rule should be applied next to a sequent stored at a proof tree node, we instead want to look at its `action`, the `maker` of any of its children.

$$\texttt{andR(1)}\;\dfrac{\texttt{closeTrue}\;\dfrac{*}{\vdash \texttt{true}} \qquad \texttt{orR(1)}\;\dfrac{\texttt{QE}\;\dfrac{*}{\vdash x \geq 0, x < 0}}{\vdash x \geq 0 \lor x < 0}}{\vdash \texttt{true} \land (x \geq 0 \lor x < 0)}$$

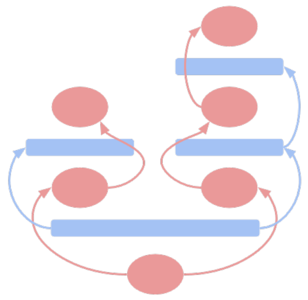


Figure 1: Derivation with corresponding proof tree (red) and execution steps (blue)

Because every single rule application has exactly one conclusion, there is a natural correspondence between proof tree nodes and execution steps. Each proof tree node has exactly one successor execution step if the node is open, and it has none if the node is closed (a $*$ in the derivation). This successor execution step is the rule application that gives the node its children.

The primary reason for this distinction is we are trying to bridge the gap between the tactic execution and the resulting derivation. The execution steps stored in the database are rule applications, and thus tactics. We store tactics in the database because we store them as we execute them. The in- memory representation of proof tree nodes corresponds more closely to the derivation. We keep track of sequents in-memory because that is what we want to display to the user. Of course, the rule information is also available, but the rules are less important than the actual sequents for viewing.

3.2 The Tactic Syntax Tree

Although a lot of the power of KeYmaera X comes from its built-in tactics that perform quantifier elimination and other differential equation reasoning, our primary focus will be on the tactic combinators. The built-in tactics are easy to handle because they cannot be split up into multiple nodes. Each built-in tactic should correspond to a single atomic node within the proof tree. Then, the naïve task of adding a node the database and updating that node when the tactic finishes or fails works as you would expect.

Instead, we will focus on the tactic combinators. I have partially reproduced the syntax table from Fulton et al. [3] for Bellerophon in Table 1.

The abstract syntax tree for these tactics is what the interpreter actually executes. Namely, it will walk over the tactics, traversing into the sequential tactic, running e_1 and then e_2; traversing into branch tactics, running e_1 on the first branch, then e_2 on the second, and so on. As such, the event stream of tactic execution starts and finishes corresponds to an Euler traversal of the tactic syntax tree. This Euler tree traversal is what we need to convert into a proof tree.

Language Primitive	Operational Meaning
$e ::= \tau$	Built-in Tactic
$\mid\ e_1\ ;\ e_2$	Sequential Composition: Applies e_2 on the output of e_1
$\mid\ (e)^*$	Saturating Repetition: Repeatedly applies e as long as it is applicable (diverging if it stays applicable indefinitely)
$\mid\ (e)^{*n}$	Fixed Repetition: Applies e for n iterations
$\mid\ <(e_1, e_2, \ldots, e_k)$	Branching: Applies e_1 to the first of n subgoals, e_2 to the second, etc.

Table 1: Meaning of tactic combinators

3.3 Conversion

To convert between these representations, we need to consider what happens to sequents during the tactic execution. As proof tree nodes are sequents, we want to produce proof tree nodes and edges to follow the sequents. In actuality, because the database stores execution steps, we want to follow the sequents in so far as they lead us to the execution steps.

The main difficulty here is the interplay between branching and sequential composition. These two tactics have almost the same syntactic representation so their Euler traversals are also almost identical. In fact, we could have changed sequential composition to take a vector of tactics as input and then their syntactic representation would only differ in the operator. Their tree traversals are identical, but their resulting conversion in the database is almost opposite.

Branching tactics have a database conversion that almost identically matches the syntax tree. If we have a branch, we want to connect each of its children to the corresponding goal in the predecessor. If you think of the predecessor of the branch tactic as the branch tactic itself, this exactly matches the syntax tree representation.

On the other hand, sequential composition works orthogonally. Rather than having each of its children connect to its predecessor, we string its children together like Christmas lights. The first child connects to the composition's predecessor and the next child connects to the first child.

The above description is slightly simplified because remember that we can sequentially compose a tactic with a branch. A branch needs multiple inputs (one for each of its children), so we need to thread multiple predecessors through each node. This gets even more complicated when you consider the fact that you can close a branch tactic at any time. This will leave a number of goals open depending on how well the proofs did during that branch.

For example, say we have a tactic like the following:

```
split2; <(
    split3; <( nil, close, nil),
    split2; <( close, nil )
); X
```

I have written split2 and split3 to stand in for built-in tactics that create 2 and 3 subgoals, respectively. Now, the X tactic will receive 3 open goals. One of its open goals will come from the partially-closed split2, and the other will come from the partially-closed split3. Moreover, the goals within the split3 were not even adjacent goals originally.

As mentioned earlier, this conversion will happen outside the sequential interpreter to separate the database concerns from core concerns of actually doing the proof. The sequential interpreter will send events over when it starts each tactic. When it finishes each tactic, it will send an event with the tactic but also the provable result (if it worked) or error (if it failed).

In order to do the transformation, we use the events sent by the sequential interpreter to build up a tree that mimics the original tactic syntax tree. In fact, we will case-analyze on the started tactic to determine the type of node. As we build up the tree, we also add the necessary nodes into the database to save our progress. See Figure 2 for an example of this conversion. There, combinator nodes are red, atoms stored in the database are blue, syntax tree edges are dashed, and proof tree edges are solid.

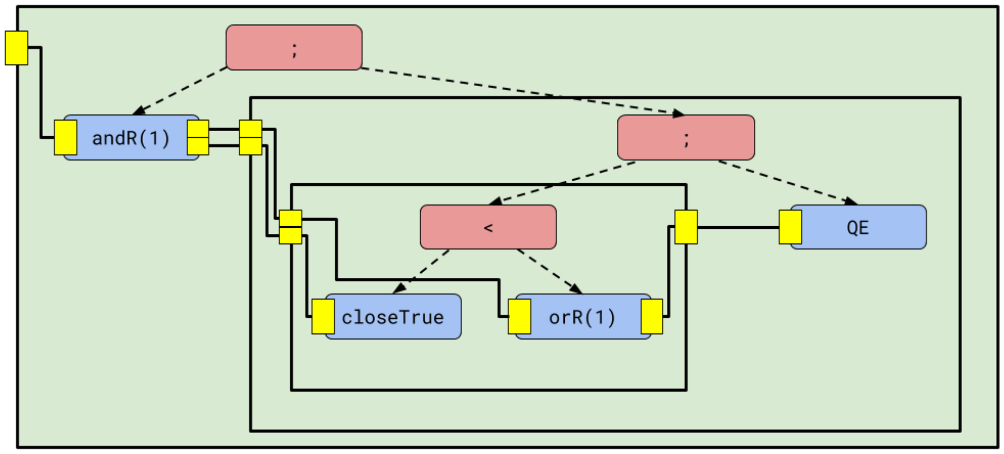


Figure 2: Syntax tree to execution step conversion

We can think of each node in the syntax tree as a box containing its subtree. Then, each box will have an incoming plug and outgoing plug. The incoming plug receives all of the edges from the predecessor box and the outgoing edge sends all edges to the successor box.

The atomic (built-in) tactics are the easiest tactics here. Atomic tactics correspond to single execution steps in the database. Because they are single execution steps, we know they must have a single edge input that will become the parent pointer in the database. After the rule application, atomic tactics produce several new subgoals. Each of these subgoals will correspond to one outgoing edge to later in the proof.

The behavior of the atomic tactics essentially informs how the other tactics should behave. We need to wire up the input plug and output plug of each combinator so that any contained atomic tactics get connected correctly. The main task is the internal wiring within a parent, between its children.

We have already described how the wiring should work for our two main tactics of concern: the branch tactic and the sequential tactic. The branch tactic splits up its input wires, one to each child. Then, the outgoing wires from each child are gathered together and fed to the output of the branch tactic.

The sequential tactic takes the entire group of input wires and feeds the group to the first child. The output from the first child is fed to the second, second to the third, until the last one is fed to the output of the sequential tactic. Although the sequential composition combinator only has two children, the saturation combinator has identical wiring behavior within.

The saturating tactic also has a wrinkle in how it executes. In particular, the saturating tactic should execute its child until the child fails or the sequent remains unaffected. If the child is a combinator that can succeed partially, we need to make sure we remove the hanging strands from the last execution. We can do this fix-up process by pruning the tree below the last successful execution of the saturating child. This only adds a small bit of extra code on top of the sequential tactic.

3.4 Pending Tactics

In addition to having this step-by-step recording, we would also like to somehow pause the execution of the tactic upon reaching an error. Rather that writing a complex back-and-forth protocol between the interpreter and the client, we can accomplish this goal by storing failed tactics as pending tactics. You can then "resume" execution by taking the tactics out of the pending wrapper, fixing the bugs, and running the new tactic. By adding pending tactics, we make the interpreter interactive. It effectively asks the user for how to proceed upon encountering each error.

Although pending tactics are much simpler than the alternative of a fully interactive interpreter, they still are a bit tricky to include. What we would like to have happen is we wrap all of the tactics after a failure point in a pending block and return that to the user. This would be possible if we were considering only sequential compositions, but the fact that we have branches means tactics in the left side of the sequential composition can fail partially. Indeed, that is almost the reason for the whole step-by-step interpreter in the first place.

To handle these issues, we store for each tactic whether it has saved failed children to the database: namely, have we written a pending tactic down for any children that did not finish. If we notice in a parent combinator if a child failed, we can add a new pending tactic for the child only if the child has not already written itself into the database.

In this process, we are trying to minimize the total number of pending blocks. Each pending block is a new block the user has to edit. For example `pending(orR(1)); pending(orR(1))` contains the same state information as `pending(orR(1); orR(1))`, but the latter only requires one edit to re-run the tactic. For this reason, we would like to defer writing a failure to the database as long as possible. By deferring, we maximize the chance that a parent combinator will also fail entirely, thus combining two otherwise separate pending blocks.

There are a few situations where this strategy does not produce the best results. For example, if we associate the sequential composition in the wrong way `(a; b); c` and tactic `b` fails, we will necessarily produce two pending blocks `(a; pending(b)); pending(c)`. The first sequential composition needs to capture the fact that `b` failed and `a` did not (because that fact will get lost by the parent), so it has to write a pending block before we can realize that `c` could also have been written there.

Again, the saturating tactic rears its head in this area. The children of a saturating tactic might produce pending blocks if they fail. Fortunately, the pruning solution from the previous section neatly handles these pending blocks as well.

We can also special-case the repeat tactic to improve its saved behavior. To write out a failure in the repeat tactic, we can write out a pending repeat tactic with the number of iterations fixed. Of course, the repeat tactic will get fully expanded eventually by the step-by-step interpreter, but that is the intended behavior. You would like to capture the sequents after each repetition, and expanding the tactic is the only way to do that.

4 User Proof Orientation

4.1 Tactic Locators

We would like to connect the tactic with the proof more closely so that the user is able to recall what they did. To do this, we need to expose some additional tactic metadata along with the simple tactic text. We would like to be able to figure out a mapping from positions within the full tactic to their corresponding proof tree nodes.

One might initially think that this mapping is computable by searching for the tactic of a certain proof node within the whole tactic. This may work for more complicated tactics that take in formulas as arguments, but many of the built-in tactics are applied all over the place. Thus, this method will not give the necessary unique results. Moreover, we were unable to determine a reasonably efficient algorithm for doing the reverse lookup. This reverse lookup problem consists of figuring out which of a set of strings appears as a substring that meets the desired lookup point.

For the reverse lookup problem, an easy solution is to maintain a mapping from each position to its corresponding node. This mapping does contain all of the necessary information to perform the forward lookup as well, but it seems unnecessarily verbose.

The solution we landed on was to store, for each node, the range within the full tactic corresponding to it. Then, we can look up the range for a particular node in constant time using a map, and we can look up the node corresponding to a point by using binary search on the sorted list of ranges.

To actually generate this information, we modify the tactic stringifier to output a sequence of tokens rather than the raw string. Tokens are either padding, providing combinators and whitespace, or identified tactics. With the token stream we can map it down into a raw string to serve as the tactic text, but we can also fold down the identified tactics into tactic locators. As we sweep through the tokens, we can keep track of our position with the resulting string fill in the location of a token each time we reach an identified tactic. The resulting pair of objects is exactly the pair of tactic text and corresponding tactic locators that we need.

4.2 Connecting Tactics and Proofs

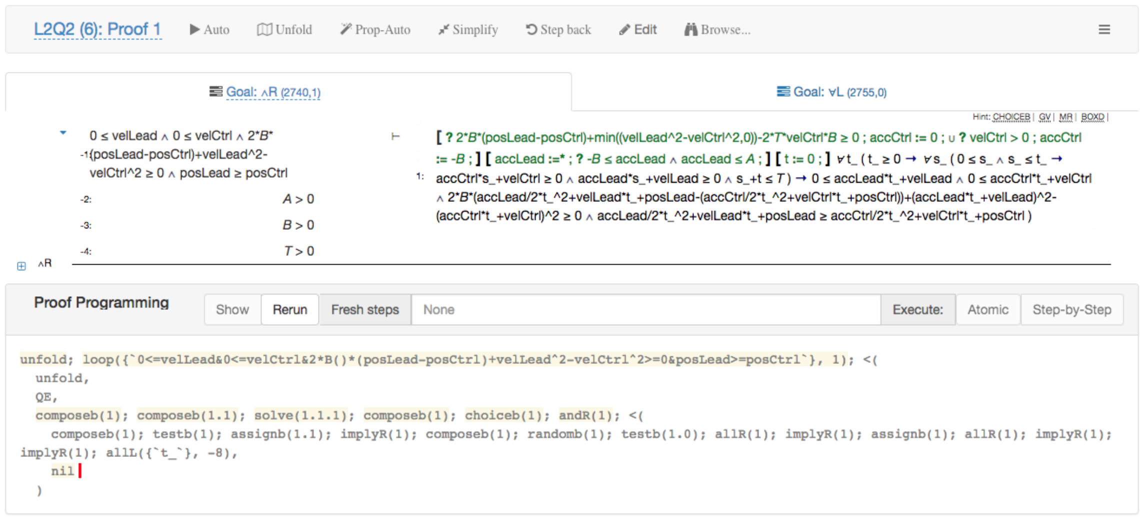
To help the user figure out where they are within the proof, we highlight the tactic text. Helping users figure out where they are within the proof is important because the process of executing tactics is opaque. The tactic executes while the user interface is paused, and then the interface abruptly jumps to the new state of the proof. You can only really figure out where each goal came from by inspecting its contents and reasoning about what the goals in certain sections of the proof should look like.

Of course, knowing where you are in the proof is not strictly necessary for completing the proof. All you need to do to complete the proof is get the current goals to close, not understand where those goals came from. But having the history in mind helps with getting a better understanding of how the proof worked, and identifying the key ideas.

Towards that end, we can identify goals with their path through the tactic. To show this path, we highlight each of the tactics on the path to the original conclusion when a goal is selected. You can figure out where you are based on how this path traces back through the branches. See Figure 3 for an example of this highlighting.

To complete this picture of where a goal came from, we need to be able to see the sequents executed in past parts of the proof. Already, the proof editor provided a way to do this. You could view the history of sequents by expanding parents one-by-one. Of course, this one-by-one business is what we have been trying to avoid this whole time.

Instead of needing to continually expand parents, the new tactic metadata allows you to click directly on a tactic and show its input sequent. In this way, we can connect the tactic text back to the sequents they operate on. Now, you can view the history all along a sequent by looking at the highlighted tactics and viewing each of their input sequents. The main value, though, is the ability to jump around the sequents. If you wanted to view



```
-1: 0 ≤ velLead ∧ 0 ≤ velCtrl ∧ 2*B*(posLead-posCtrl)+velLead^2-velCtrl^2 ≥ 0 ∧ posLead ≥ posCtrl
-2: A > 0
-3: B > 0
-4: T > 0
⊢
1: [ ? 2*B*(posLead-posCtrl)+min((velLead^2-velCtrl^2,0))-2*T*velCtrl*B ≥ 0 ; accCtrl := 0 ; ∪ ? velCtrl > 0 ; accCtrl
:= -B ; ] [ accLead :=* ; ? -B ≤ accLead ∧ accLead ≤ A ; ] [ t := 0 ; ] ∀t_ ( t_ ≥ 0 → ∀s_ ( 0 ≤ s_ ∧ s_ ≤ t_ →
accCtrl*s_+velCtrl ≥ 0 ∧ accLead*s_+velLead ≥ 0 ∧ s_+t ≤ T ) → 0 ≤ accLead*t_+velLead ∧ 0 ≤ accCtrl*t_+velCtrl
∧ 2*B*(accLead/2*t_^2+velLead*t_+posLead-(accCtrl/2*t_^2+velCtrl*t_+posCtrl))+(accLead*t_+velLead)^2-
(accCtrl*t_+velCtrl)^2 ≥ 0 ∧ accLead/2*t_^2+velLead*t_+posLead ≥ accCtrl/2*t_^2+velCtrl*t_+posCtrl )
```

```
unfold; loop({`0<=velLead&0<=velCtrl&2*B()*(posLead-posCtrl)+velLead^2-velCtrl^2>=0&posLead>=posCtrl`}, 1); <(
  unfold,
  QE,
  composeb(1); composeb(1.1); solve(1.1.1); composeb(1); choiceb(1); andR(1); <(
    composeb(1); testb(1); assignb(1.1); implyR(1); composeb(1); randomb(1); testb(1.0); allR(1); implyR(1); assignb(1); allR(1); implyR(1);
implyR(1); allL({`t_`}, -8),
    nil
  )
```

Figure 3: KeYmaera X with tactic highlighting

the entire trace, you could already do that by expanding.

5 Execution-Step Tactic Patching

In order to enable all of the new features, we need to have efficient tactic editing. All of the previous work with adding pending tactics requires lots of editing to undo the pending tactic, fix up the broken tactics, and apply the new desired outcome. To that end, it is infeasible to delete the entire proof and rewind from the original sequent in order to apply an edited tactic.

This would be especially infeasible if there were many computationally- intensive tactics towards the beginning of a proof. Unfortunately, that is exactly the situation with many proofs. After unfolding all of the bookkeeping and applying a couple of loop invariants, you reach some differential equations. Then, you add several differential invariants by cutting them in before working through the rest of the proof. Right at the top of your proof, you could have several differential invariant computations and quantifier eliminations to handle the loop premises.

To prevent this slow situation, we need to avoid pruning the proof tree all the way down to the root. Instead, we would like to prune the minimal possible node: the least common ancestor of all the changes. By pruning as high in the tree as possible we minimize the number of re-executions.

Once we have found the least common ancestor, we can prune at that execution step and re-run the tactic below it. As we already have infrastructure for running a tactic below a node—that is the primary way that new steps are added to a proof—the main difficulty is determining what is the least common ancestor and what tactic lives below it.

The existing tactic diff-and-patch tools will not work for this task because we do not consider tactics only by their syntax tree. We instead need to consider tactics by their execution steps. The difference tool telling us there is a small difference at a certain point within the tactic is not useful that point occurs within a single execution step. Instead, we need to do an asymmetric diff-and-patch that compares the proof tree with the input tactic.

To make the process slightly easier, I extended the tactic stringifier from the previous part to wrap single steps in parentheses. That way, the tactic sent to and editing by the client would automatically group atomic tactics as a single tactic. Without this, sequential compositions would be parsed out of order and the analysis would need to normalize across sequential compositions.

The final algorithm recursively analyzes the current tactic. Because we know how the execution steps are serialized from the database, we can determine what execution step is expected depending on the current tactic. If the tactics match, we can pass off to the children of the tactic and the children of the step. If the tactics differ, we can return the current node and tactic as the first point of difference. Because we want to have a single unified point of difference, if more than one child reports back positive, we need to also return the current node instead.

By recursively comparing the tactic with the execution step, and with some clever stringification, we are able to figure out the point and tactic to apply to edit. With this method, fixing up a pending tactic does not actually require re-applying any old tactics. This gives us the “interactive” interpreter behavior that we were looking for.

6 Related Work

The Bellerophon paper introduces the tactic language and touts one of its main benefits as being able to specify heuristic search automation tasks [3]. Indeed, Bellerophon does provide combinators for mixing together built-in tactics to enable heuristic search over proofs. However, this perspective overstates the benefits of automation. Indeed, Fulton et al. [3] even acknowledge that a key component of proving properties about hybrid systems is human intervention to provide assistance for finding invariants and deciding real arithmetic.

If the prover is hostile towards the user, having lots of automation will not prove all of the properties. Unfortunately, the semidecidability of hybrid systems formulas means that we need the human intervention to find the necessary invariants and (in practice) real arithmetic simplifications. Fulton et al. [3] mention three key components of a good hybrid-system prover: small, trusted core, a library of high-level primitives for common tasks, and scriptable heuristic search automation. A fourth key component is an effective interface for making use of the other three.

The Coq IDE Coqoon provides a different approach to proving theorems [2]. Rather than determining what tactics to run as the proof progresses, you can write down a full tactic script and then execute that script step-by-step. As you execute the script, any errors that you find can be immediately edited and re-run. In this way, Coqoon behaves very much like the new step-by-step listener. However, Coq and other more general- purpose theorem provers like Isabelle [1] do not provide the same specialization to the task of proving hybrid systems as KeYmaera X [4]. KeYmaera X has a unique suite of tactics adapted to this domain.

7 Conclusion and Future Work

This work provides new tactic and execution tools to the KeYmaera X theorem prover. These tools improve the proof experience for users because they limit the amount of punishment users receive for writing down an incorrect tactic. Rather than forcing the tactic to restart, users can engage in a dialogue with the prover, editing and re-running tactics until they are accepted. With decreased penalties to failure, users will be free to experiment more and potentially discover better pathways to proofs.

The new tools also more closely couple the current proof state with its tactic. By providing a highlighted connection between the active goal and how it came about, users can gain more holistic understanding of their proofs. Conversely, users can verify their understanding by peeking back at the sequents exposed by past executions of tactics.

To further improve these tools, we could investigate a more permissive database format that allows convergent tree branches. This would make tactic idioms for formulating and proving a lemma valid in the proof tree. In addition, even though the new tactic editing prunes at the least common ancestor of an edit, tactics after the edit might remain unchanged. Maybe we can further decrease edit time by somehow capturing the work done by these previous tactics and applying it to the changed sequent without having to re- run them.

References

[1] Bruno Barras, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, pages 359–363, 2013. doi: 10.1007/978-3-642-39320-4_29. URL `https://doi.org/10.1007/978-3-642-39320-4_29`.

[2] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. Coqoon -

an IDE for interactive proof development in coq. *STTT*, 20(2):125–137, 2018. doi: 10.1007/s10009-017-0457-2. URL `https://doi.org/10.1007/s10009-017-0457-2`.

[3] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. Bellerophon: Tactical theorem proving for hybrid systems. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 207–224, 2017. doi: 10.1007/978-3-319-66107-0_14. URL `https://doi.org/10.1007/978-3-319-66107-0_14`.

[4] Stefan Mitsch and André Platzer. The keymaera X proof IDE - concepts on usability in hybrid systems theorem proving. In *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016.*, pages 67–81, 2016. doi: 10.4204/EPTCS.240.5. URL `https://doi.org/10.4204/EPTCS.240.5`.